A Gentle Introduction to Clean

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

(based on slides by Pieter Koopman)

What this lecture is about

- You know how to program in a functional programming language[1]
- Get you familiar with the notation, concepts, and useful programming skills in Clean
- Clean has been installed[2]
- Clean history
  - 1987: initial design as intermediate language for efficient functional language compilation
    - (parallel) term graph rewriting, strictness annotations, distribution annotations, (P)ABC-machine
  - 1995: Clean moves from intermediate language to front-end language (versions 1.x)
    - uniqueness types, GUI support with Object I/O
  - 2000: Clean moves to version 2.x
    - proof assistant (Sparkle), dynamic types, generic programming, model based testing (G∀st)
  - 2007…: Task Oriented Programming (iTask, mTask, Tonic, TopHat)

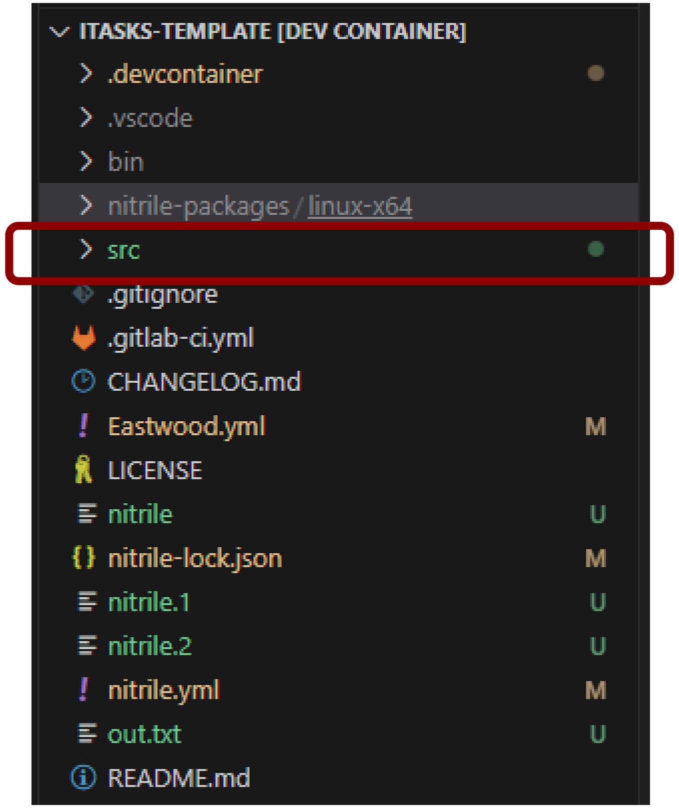
---

[1] Brightspace:Getting prepared:Introductory course functional programming in Clean

[2] https://gitlab.com/clean-and-itasks/itasks-template

Clean files

- `prog.icl` implementation module, function definitions
- `prog.dcl` definition module, the exported definitions list only the signatures
- `prog.abc` generated abstract machine code
- `prog.o` object code, generated machine code
- `prog` executable



- The main `.icl` module does not need a `.dcl` file
  - the first line is: **`module`** *`filename`*
- Any other implementation module:
  - **`implementation module`** *`filename`*

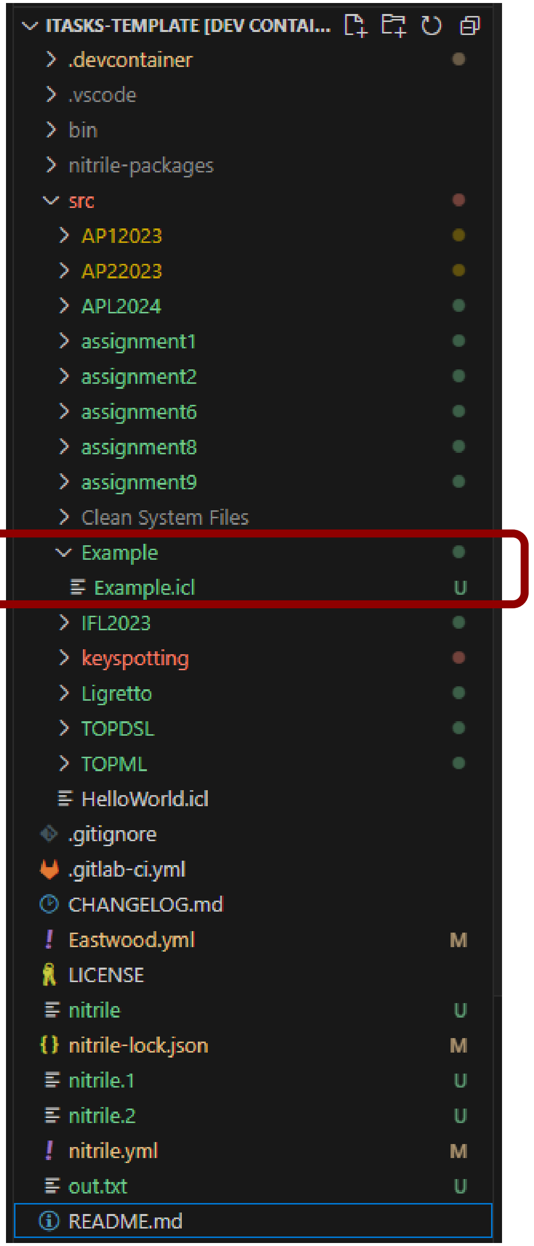
Clean initial expression

- Any Clean program starts evaluating `Start`
- Example: file `Example.icl`

```
module Example
import StdEnv

fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

Start :: Int
Start = hd (map fac [3,1,4,1,5])
```

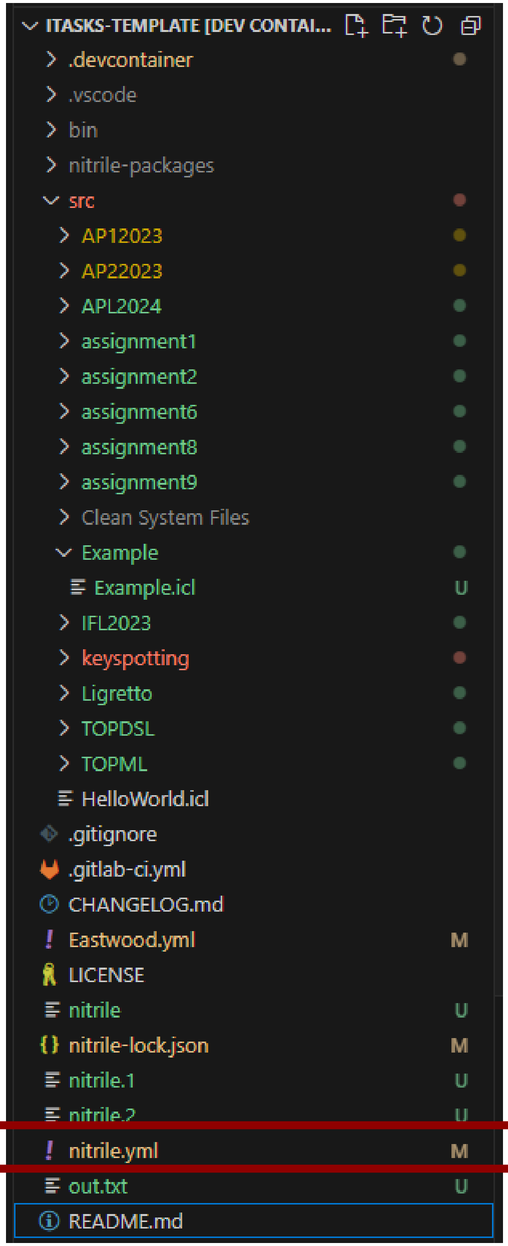


Clean initial expression

- Any Clean program starts evaluating `Start`
- Example: file `Example.icl`

```yaml
clm_options:
  compiler: cocl-itasks
  fusion: GenericFusion
build:
  application:
    script:
      - clm:
        src: [src/Example/]
        main: Example
        target: bin/Example/Example
        bytecode: prelinked
        heap: 20m
        generate_descriptors: true
        export_local_labels: true
        strip: false
        post_link: web-resource-collector
```

compiler path to main module

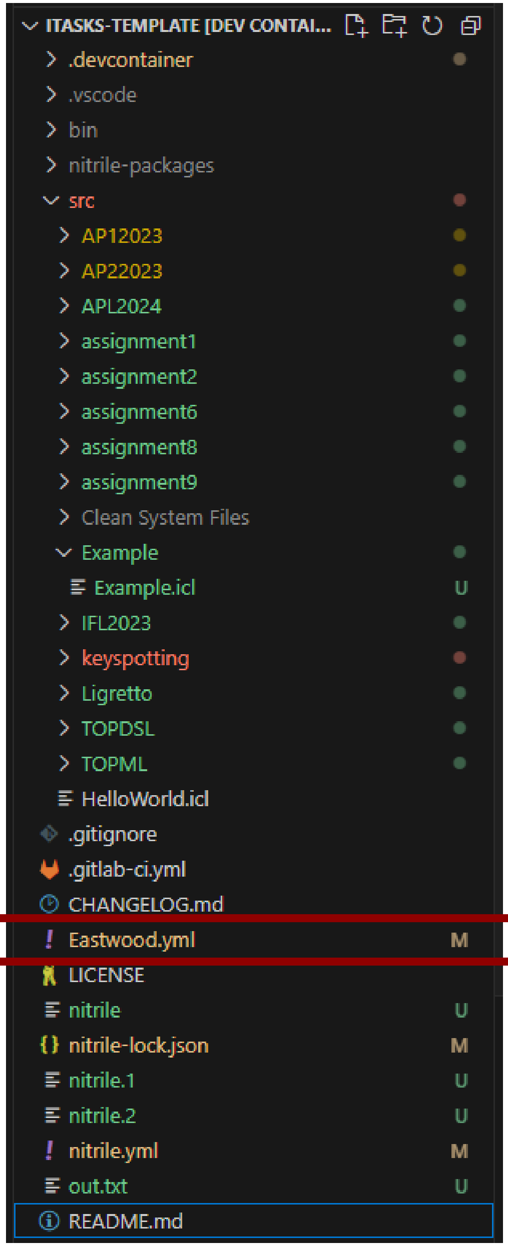


Clean initial expression

- Any Clean program starts evaluating `Start`
- Example: file `Example.icl`

```yaml
compiler: cocl-itasks
paths:
  - src        Gijs Alberts, 24 months ago • Add Eastwood.yml
  - src/Example
  - nitrile-packages/linux-x64/abc-interpreter/lib
  - nitrile-packages/linux-x64/base-stdenv/lib
  - nitrile-packages/linux-x64/clean-platform/lib
  - nitrile-packages/linux-x64/gast/lib
  - nitrile-packages/linux-x64/graph-copy/lib
  - nitrile-packages/linux-x64/itasks/lib
  - nitrile-packages/linux-x64/tcpip/lib
```

display compiler issues in source code



Clean initial expression

- Any Clean program starts evaluating `Start`
- Example: file `Example.icl`

```
module Example
import StdEnv

fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)

Start :: Int
Start = hd (map fac [3,1,4,1,5])
```

module name must match file name

standard library

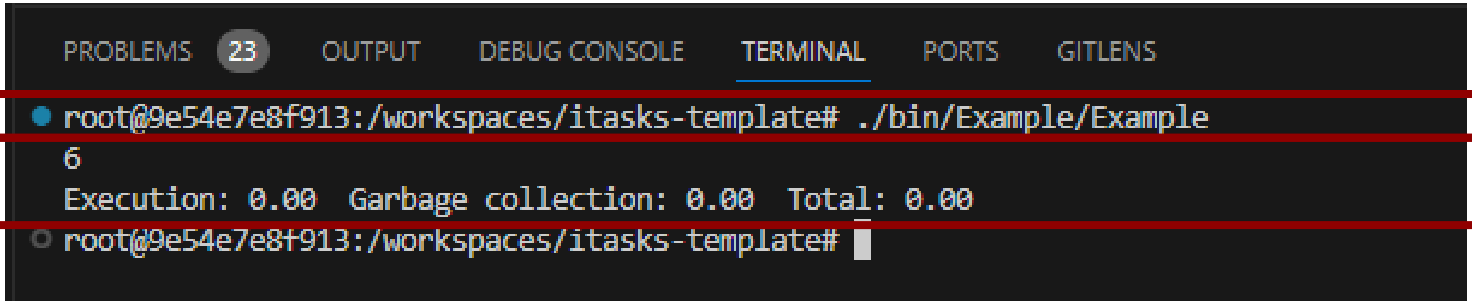
it is encouraged to specify types

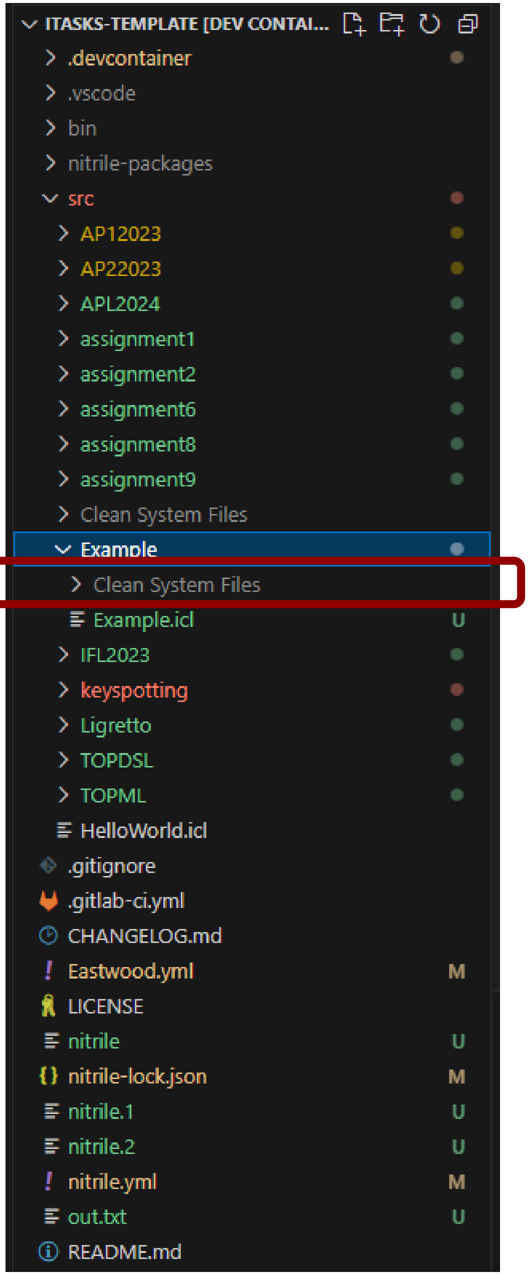
your definitions

program evaluates `Start` and prints result

Clean initial expression

- Any Clean program starts evaluating `Start`
- Example: file `Example.icl`
- In VS Code Terminal:
  - Compile with `nitrile build`
  - Run executable





Algebraic Data Types (ADT)

```
:: Colour = Black | White

invert :: Colour -> Colour
invert Black = White
invert _     = Black

:: Maybe a = Just a | Nothing

:: List a = .: | (:.) infixr 1 a (List a)

my_length :: (List a) -> Int
my_length .:       = 0
my_length (_ :. l) = 1 + my_length l
```

We can have any positive number of constructors

We typically use

```
:: ? a = ?Just a | ?None
```

and

```
:: [a] = [] | [a:[a]]
```

Records (structs)

```
:: Point
 = { x :: Int
   , y :: Int
   }

origin   = {x = 0, y = 0}

invert p = {x = ~p.x, y = ~p.y}

setX a p = {p & x = a}

setY a p = {Point|p & y = a}

getX {x} = x

getY p = p.y
```

We can have any numbers of fields

Like ADTs we can have any number of type variables

```
:: BM a b = {ab :: a -> b, ba :: b -> a}
```

add the type constructor name if the compiler cannot decide the type based on the field names

also for member selection:

```
getX {Point|x} = x
getY p = p.Point.y
```

Macro

- A macro is a definition expanded at compile time
  - function types are not allowed here

```
(o) infixr 9 //:: (b -> c) (a -> b) -> a -> c
(o) f g :== \x = f (g x)

($) infixr 0 // :: (a -> b) a -> b
($) f :== f
```

```
(o) infixr 9 :: (b -> c) (a -> b) -> a -> c
(o) f g = h where h x = f (g x)
```

infix operators are functions with 2 arguments

- More efficient code
- Compile time evaluation: no recursion
- Also for types (aka synonym types):

```
:: Pair x y :== (x, y)
```

Type Constructor Classes (overloading)

- A type constructor class is a set of different functions with the same name
- Types are used to distinguish these functions

```
class nat a
where add  :: a a -> a
      null :: a
```

- Adding an instance to the type constructor class

```
instance nat Int
   where add x y = x + y
         null = 0
```

```
:: N = Z | S N

instance nat N
   where add Z y = y
         add (S n) m = S (add n m)

         null = Z
```

- Shorthand syntax if a type constructor class has only one member function

```
class (+) infixl 6 a :: !a !a -> a
```

Type Constructor Classes (overloading)

- Type constructor class variables can have any kind

```
class stack s
where push  :: a (s a) -> s a
      pop   ::   (s a) -> s a
      top   ::   (s a) -> a
      empty ::   (s a) -> Bool
instance stack []
   where push e stack = [e:stack]
         pop [e:rest] = rest
         top [e:rest] = e
         empty stack  = isEmpty stack
```

s gets an argument:
s has kind * -> *

[a] has kind *
[ ] has kind * -> *

Using Type Constructor Classes (overloading)

- A function can work for many types
- Example: `sum` works for any type `a` with:
  - an operator `+`, and
  - a constant `zero`

```
sum :: [a] -> a | +, zero a
sum []      = zero
sum [a : x] = a + sum x
```

- Rational numbers:

```
:: Q
 = {q :: !Int, n :: !Int}

instance zero Q
   where zero = {q = 0, n = 1}
instance  + Q
   where (+) x y = norm {q = x.q*y.n + y.q*x.n
                        ,n = x.n*y.n
                        }
norm :: !Q -> Q
norm {q, n} = {q = q / x, n = n / x}
where x = gcd q n
```

List

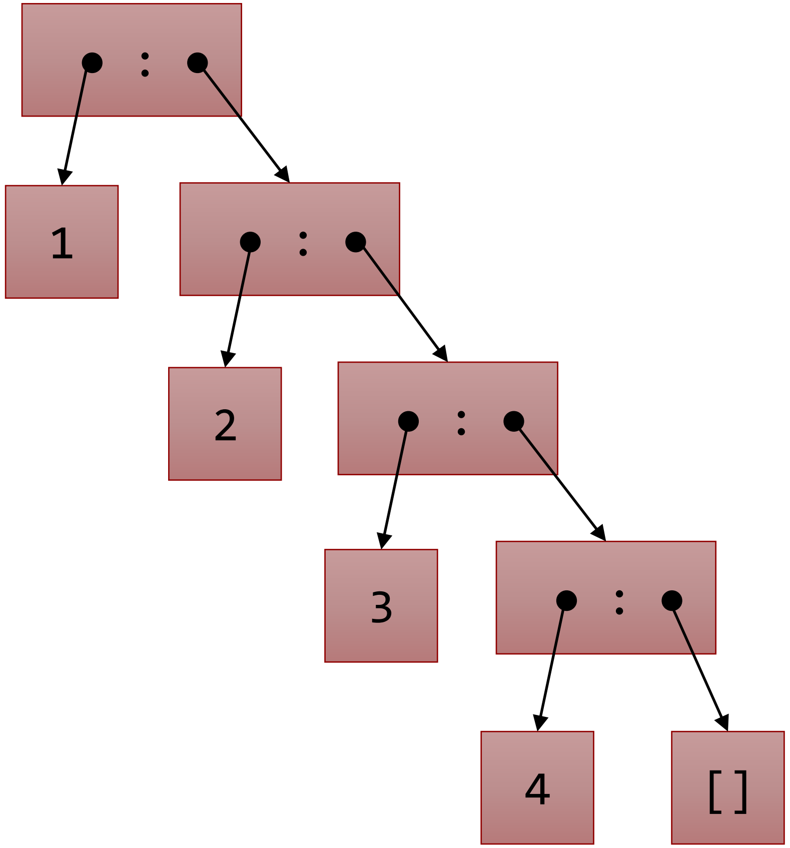
- Just special syntax for

```
:: List a = Nil | Cons a (List a)
```

- Notations

```
[1:[2:[3:[4:[]]]]]
[1:2:3:4:[]]
[1,2,3,4]
```

- `['hi']` is shorthand for `['h','i']`



List enumerations

```
class IncDec a | +, -, one, zero a
where inc x :== x + one
      dec x :== x - one

class Enum a | <, IncDec a

[1..4]          → [1,2,3,4]

[0.5..3.8]      → [0.5,1.5,2.5,3.5]

['a'..'c']      → ['abc']

[1,3..10]       → [1,3,5,7,9]

[10,8..2]       → [10,8,6,4,2]

[1..]           → [1,2,3,4,5,6…
```

List comprehensions

- generator `[ …x…    \\ x <- xs]`
- product `[ …x…y… \\ x <- xs , y <- ys]`
- pairs `[ …x…y… \\ x <- xs & y <- ys]`
- filter `[ …x…    \\ x <- xs | pred x]`

similarly we have array comprehensions

`[x*x    \\ x <- [1..4]]`	`→ [1,4,9,16]`
`[x*x+1 \\ x <- [1..4] | isOdd x]`	`→ [2,10]`
`[(x,y) \\ x <- [1,2] , y <- ['abc']]`	`→ [(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]`
`[(x,y) \\ x <- [1,2] & y <- ['abc']]`	`→ [(1,'a'),(2,'b')]`

Patterns, top-level guards, case distinctions

```
hd :: [a] -> a
hd [x:_]  = x
hd _      = abort "hd of []"

take :: Int [a] -> [a]
take n l
| n <  0  = abort "take of negative"
| n == 0  = []
take n [x:xs]
          = [x : take (n-1) xs]
take n [] = []
```

```
hd list = case list of
            [x:_] = x
            _     = abort "hd of []"
```

```
take n l = if (n <  0) (abort "take of negative") (
           if (n == 0) [] (
             case l of
               [x:xs] = [x : take (n-1) xs]
               _      = []
           ))
```

Array[1]

- Arrays are like records with integers as field names
  - All fields have the same type
  - Length is finite and fixed
- Arrays come in different flavours:
  - Lazy: elements are evaluated by need
  - Strict: elements are always evaluated
  - Unboxed: elements are always evaluated and stored in the array (instead of a reference)
- Efficient in-place array updates with uniqueness types
- Strings are likely the only arrays you really need in Clean

```
:: String :== {#Char}
```

- SaC is much more sophisticated in array manipulations

---

[1] Appendix gives a complete example with arrays

Modules

good practice to minimize exported symbols (avoid name clashes)

```
implementation module Bin
import StdEnv

ins :: a (Bin a) -> Bin a | < a
ins a Leaf   = Bin Leaf a Leaf
ins a (Bin l b r)
| a < b       = Bin (ins a l) b r
| otherwise = Bin l b (ins a r)

inorder :: (Bin a) -> [a]
inorder Leaf          = []
inorder (Bin l a r) = inorder l ++ [a: inorder r]
```

```
definition module Bin
import StdOverloaded

:: Bin a
 = Leaf
 | Bin (Bin a) a (Bin a)

ins     :: a (Bin a) -> Bin a | < a
inorder ::   (Bin a) -> [a]
```

```
module BinDemo
import Bin, StdEnv

mysort :: [a] -> [a] | < a
mysort l
      = inorder (foldr ins Leaf l)

Start = mysort ['u','o'..'a']
```

definition of `Bin` does not have to be repeated, but the types of the exported functions have to be repeated

`['ciou']` ←

Generics (deriving)

- For any non-synonym type `T`:

```
import Data.GenEq
derive gEq T                // gives you === and =!=, both :: !T !T -> Bool
import Data.GenLexOrd
derive gLexOrd T            // gives you =?= :: !T !T -> LexOrd
:: LexOrd = LT | EQ | GT // gEq is defined for LexOrd
import Data.GenFDomain
derive gFDomain T           // gives you the list of all values of type T
import Text.GenPrint
derive gPrint T             // gives you printToString :: !T -> String
import Text.GenParse
derive gParse T             // gives you parseString :: !String -> ?T
                            //       and parseFile   :: !File   -> ?T
```

```
:: N = Z | S N
derive gEq      N
derive gLexOrd  N
derive gFDomain N
derive gPrint   N
derive gParse   N
```

"bimap_ss" no instance available of type […] [T]

```
derive bimap []
```

```
  Z === S Z                              → False
S Z === S Z                              → True
  Z =!= S Z                              → True
S Z =!= S Z                              → False
  Z =?= S Z                              → LT
S Z =?= S Z                              → EQ
take 5 list                              → [Z,S Z,S (S Z),S (S (S Z)),S (S (S (S Z)))]
printToString  Z                         → "Z"
printToString (S (S Z))                  → "S (S Z)"
parseString "Z" === (?Just Z)            → True
parseString "S (S Z)" === (?Just (S (S Z)))  → True

list :: [N]
list = gFDomain{|*|}
```

More information

- https://cloogle.org/
- https://cloogle.org/doc/
- https://top-software.gitlab.io/clean-lang/
- https://clean.cs.ru.nl/images/3/36/ConciseGuideToClean3xStdEnv.pdf
- https://clean.cs.ru.nl/Functional_Programming_in_Clean
- your teachers
- most Haskell material will work with minor adaptions
- Clean for Haskell Programmers (Brightspace)

Appendix: Game of Life with Clean arrays

```
LIVE     :== '*'
DEAD     :== ' '
:: Coord :== (Int,Int)
:: Cell  :== Char

universe :: !Int !Int -> *{#*{#Cell}}
universe m n = {# {#DEAD \\ _ <- [1..n]} \\ _ <- [1..m]}

instance toString {#{#Cell}}
   where toString u
            = join "\n" ([h] ++
                         ['|' <+ str <+ '|' \\ str <-: u] ++
                         [h])
         where (_,no_cols) = dimension u
               h           = createArray (2+no_cols) '-'

dimension :: !{#{#Cell}} -> (!Int,!Int)
dimension u = (size u, size (u.[0]))
```

```
neighbours :: !Coord !{#{#Cell}} -> Int
neighbours (r,c) u
   = length (filter ((==) LIVE) [u.[r-1].[c-1],u.[r-1].[c],u.[r-1].[c+1]
                                ,u.[r  ].[c-1],            u.[r  ].[c+1]
                                ,u.[r+1].[c-1],u.[r+1].[c],u.[r+1].[c+1]
                                ])

next :: !{#{#Cell}} -> *{#*{#Cell}}
next old
   = cells old (universe no_rows no_cols) [(r,c) \\ r <- [1..m-2]
                                                  , c <- [1..n-2]]
where (m, n) = dimension old

cells :: !{#{#Cell}} !*{#*{#Cell}} ![Coord] -> *{#*{#Cell}}
cells old new [] = new
cells old new [(r,c):cs]
   = cells old {new & [r].[c] = cell old.[r].[c] (neighbours (r,c) old)} cs
```

Appendix: Game of Life with Clean arrays

```
cell :: !Cell !Int -> Cell
cell LIVE n = if (n == 2 || n == 3) LIVE DEAD
cell dead n = if           (n == 3) LIVE DEAD

glider :: !Coord !*{#*{#Cell}} -> *{#*{#Cell}}
glider (r,c) u
   = {u & [r  ].[c-2] = LIVE, [r  ].[c-1] = LIVE, [r  ].[c] = LIVE
        , [r+1].[c-2] = DEAD, [r+1].[c-1] = DEAD, [r+1].[c] = LIVE
        , [r+2].[c-2] = DEAD, [r+2].[c-1] = LIVE, [r+2].[c] = DEAD
     }

Start = map (\u = u <+ '\n') (take 5 (iterate next u0))
where
   u0 = glider (5,5) (universe 10 10)
```

```
------------   ------------   ------------   ------------   ------------
|          |   |          |   |          |   |          |   |          |
|          |   |          |   |          |   |          |   |          |
|          |   |          |   |          |   |          |   |          |
|          |   |          |   |          |   |          |   |          |
|          |   |    *     |   |    **    |   |    **    |   |    ***   |
|   ***    |   |    **    |   |   * *    |   |     **   |   |      *   |
|     *    |   |   * *    |   |     *    |   |    *     |   |     *    |
|    *     |   |          |   |          |   |          |   |          |
|          |   |          |   |          |   |          |   |          |
|          |   |          |   |          |   |          |   |          |
------------   ------------   ------------   ------------   ------------
```