Clean more like I wanna clill myself

A guide to programming in Clean
without the suicidal tendencies.

[Contents]

click on it

Authors: Melle P. Starke & Crippling D. Pression

1617 Functional Programming for Artificial Intelligence Students
(NWI-IBC015-2016-KW3-V)

Radboud University, Netherlands
Faculty of Social Sciences
Artificial Intelligence

27 March 2017

"SWEET JEZUS WHY!?"

If you're reading this, chances are that either your arms are suffering from anemia from waiting for the TA's to finish helping the 20 people in the room, or you're trying to program in Clean without any TA's whatsoever, and you just wanna end it all. Fret not! This document serves as an encyclopedia for basic explanations of clean code and datatypes, helpful links and forbidden coding secrets whose names none dare speak.

Contents

1 Things Everyone Should Know

Clean can be quite confusing and illogical, so here's some tips on how to get around this dumpster fire:

- **In order to work on a project**, the .prj file, the .icl file (this is where you'll be coding in) and the .dlc file need to be accessible by clean.
- **When wanting to run an .icl file** you first need to load the .prj file.
- **You can only have one .prj file open at a time**, but you can have multiple .icl files open.
- **The .icl file can be loaded by opening the .prj file** and double-clicking it in the light brown coloured project window.
- **You can open de .dcl file** by pressing Ctrl + / (also works the other way around).
- **If you don't know what a certain type or function does,** you can look it up by Ctrl + double-clicking it.
- **If you disable a function in the .icl file** by commenting it out ("//"), you might need to do the same thing in the .dcl file.
- **You can comment multiple lines** by putting "/*" at the top and "*/" at the bottom.
- **Disable "info" in the pink "Errors & Warnings" window.** It'll only scare you.
- **Clean's Ctrl+Z command initially only stores one change at a time.** But apparently there is a way to fix it. If you do delete your entire code and can't get it back, just load the .icl file again, there's no autosave.
- **You can add line numbers** by going to Defaults → Window Settings → Editor Settings → Show LineNrs.
- **There's a lot of helpful commands in Clean's standard environment** (StdEnv). Seriously, if you're stuck, check the StdEnv section in Functional Programming in Clean first (see Helpful Links).
- **You can close the .exe file by pressing any key.** Keeping it open will prevent you from running the code.
- In clean, `::` **is always used to assign types, and** `=` **is always used to assign values.**
- **If you can't find something in this guide** look for it in Helpful Links or just Google the Haskell code, it has really similar syntax.

2 Syntax

2.1 Differences Compared to Java

Clean is a functional programming language. This means it needs to execute code in "one line". Code such as

```
if(raining){
    getCoat(waterProof);
    getUmbrella();
}
```

won't work. You cannot group commands together. Because of this, you also cannot easily save some of your values, then come up with the next step, it all needs to be done in *one* step.

Clean does not have while-loops or for-loops. Instead, you use recursion when you need to loop something. It also does not show syntax errors in real-time, you have to bring it up to date (Ctrl+U) in order to see the errors. And whereas Java functions can only return a single value, Clean functions can return multiple, if written as Tuples. In Clean, parameters are passed to functions with spaces. If a function needs 2 arguments, it's just gonna look for the next 2 things separated by spaces. But you can group things together by using parentheses. In Clean, parentheses are used to group things together, rather than to pass parameters. Which is more like a math kind of approach.

2.2 Operators

Clean operators are rather similar to Java operators, but since there are quite a few differences, here's a list of 'em. Any operator that isn't in this list is just the same as in Java.

Operator	Output	Meaning
`zero`	`Assign`	`zero value of datatype`
`one`	`Assign`	`one value of datatype`
`x +++ y`	`String`	`"add string x to string y"`
`x ++ y`	`List`	`"add list x to list y`
`x^y`	`Int/Real`	`"x to the power y"`
`~x`	`Int/Real`	`"switch sign of x" (+ / -)`
`xs % (x,y)`	`String/List`	`slice from index x to index y of xs`
`x rem y`	`Int/Real`	`modulo (remainder of a division)`

Note: `zero` and `one` assign a value to a certain type and can be used on all basic types. `zero :: Int = 0`, for example. And `zero :: Real = 0.0`.

3 Datatypes

3.1 Basic Types

Clean does not have all the same basic datatypes as Java. So here's a list:

Name	Meaning	zero value
Int	Whole Number	0
Real	Decimal Number	0.0
Bool	Boolean	
Char	Character	''
String	String	""
a	Any Type	

NOTE: `a` is used in function input & output declaration, and can be any letter. If you use different letters in your function definition, it means that they need to be different types.

3.2 Algebraic Type

An algebraic type is simply an entirely new type, with a certain amount of possible values (so not a data structure like tuples or record types). If you know what an enumeration is in Java, it's basically that. Every basic type in Clean is an algebraic type. The definition of `Bool`, for example, looks like this:

```
:: Bool = True | False
```

With the possible values (here known as *data constructors*) separated by guards.

If you make a new algebraic type, you can of course use it in your code:

```
:: Day = Mon Tue | Wed | Thu | Fri | Sat | Sun

isWeekend :: Day -> Bool
isWeekend Sat = True
isWeekend Sub = True
isWeekend _ = False
```

This also means that `Sat :: Day`.

Apart from being separate values for `Day`, they're nothing more. They're not linked to their positions and so cannot be called with a number. `Day` is just a new type that can have 7 separate values. If you do want to link their values to certain positions, just make a list and put all the values in there in a certain order, then use `hd`, `tl` and `[x:xs]` to go through the elements.

Also, any value can only be of one type. Which means that `:: Weekend = Sat | Sun` and `:: Dice = 1 | 2 | 3 | 4 | 5 | 6` will result in an error. Also for some reason, values of a new algebraic type need to begin with a capital letter.

3.3 Data Structures

3.3.1 Tuples

A tuple is a type that consist of multiple types. There’s different kinds of tuples, like a double: (Rain,Sun) and a triple: (1,7,3). Note that singles don’t exist, a (7) is just read as an `Int`.

The types within a tuple can be anything (even another tuple) and can be different from each other. For example, `(("Rain","Sun"),1,’k’,Contains)` is a valid quadruple, with type `((String, String), Int, Char, (Char String -> Bool))`, in that order.

Tuples can be used to make a function have multiple outputs by just writing the output as a tuple. Ex:

```
toDouble ::  a a -> (a,a)
toDouble x y = (x,y)
```

NOTE: The order of the elements in a tuple is highly important. Each tuple with a certain order and amount of types is a distinct type of its own.

Defining a new tuple in Clean looks like this:

```
Tuple1 ::  ((String,String),Int,Char)
Tuple2 ::  (Bool,Real,Char String->Bool)
```

NOTE: Tuples don’t always have to be defined. `Start = (1,7,3)` is perfectly fine. But sometimes it can be helpful to define a tuple. And when writing a function definition that uses a tuple it’s of course necessary to define the type of the tuple.

3.3.2 Record Types

A record type is a type consisting of multiple types. Record types can contain different sorts of types, much like Tuples. But unlike tuples, the order of the elements is irrelevant (even when defining and constructing the record type). This is because you name each element (a bit like object oriented programming in Java):

```
::  Person = { name ::  String
             , birthdate ::  (Int,Int,Int)
             , parents ::  (Person,Person)
             }
```

:: signifies new type

”name :: type”, separated by commas

} signifies record type

NOTE: This can be written in one line as well, but this way comments look better (And for some reason elements of a record type need to begin with a lower case letter).

This is only the definition of the new record type `Person`. It does not contain any values yet. In order to make a new `Person`, you have to give it a name and assign values to it:

```
Bert :: Person
Bert = { name = Bert
       , parents = (Jochem,Greta)
       , birthdate = (30,2,1989)
       }
```

define Bert as type Person

notice similar structure. But with = instead of ::

Getting Record Type Values

When wanting an element of a record type, you generally use dots (much like in Java when you want a value or function of an object).

```
:: Birthdate = {day :: Int, month :: Int, year :: Int}

getYear :: Birthdate -> Int
getYear x = x.year
```

The above function returns the element `year` of the record type `Birthdate`. As for an example:

```
CuriousBetsy's :: Birthdate
CuriousBetsy's = {Day = 2, month = 2, year = 1984}
Start = getYear CuriousBetsy's
```

This function will have as output `1984`.

Record types can be passed in 2 ways. with a dot, or with curly brackets. The function `getYear` might just as well have been `getYear {year} = year`. In the latter case you don't need to pass all the elements, just the ones you need. If you do need more you can separate them with commas.

3.3.3 Lists

Lists, like tuples, are data structures with elements on certain positions. But unlike tuples, a list can contain zero or more elements and can only contain elements of the same type.

There are a few ways to write the contents of a list:

canonical:	`[1:[2:[3:[]]]]`
shorthand canonical	`[1:2:3[]]`
plain	`[1,2,3]`
enumeration	`[1..3]`

The last one is more of a way to make a list. This particular one creates a list of all numbers between and including 1 and 3, with intervals of 1. So: `[1,2,3]`. You can change the interval of an enumeration by specifying the 2nd element. It'll enumerate all numbers between and including the 1st and last number, with intervals of the 2nd number minus the 1st. So `[1,3..9]` = `[1,3,5,7,9]`. If the last number is not in the list, it'll go up to the last included element before that. So `[1,3..10]` = `[1,3,5,7,9]`. And infinite lists are also possible: `[1..]`.

There's a few symbols that you can use for lists in pattern definitions:

Symbol	Example	Meaning
`:`	`[x:xs]`	splits head (1st element (x)) from tail (the rest (xs))
`,`	`[x,y,z:xs]`	"these consecutive elements"
`_`	`[x:_]`	"I don't care what this is"
`[]`	`[x:[]]`	empty list

This also means that a list with one element can be written as `[x:[]]`. Also note that in the case of `[x:xs]` and `[x:_]`, both tails can be empty.

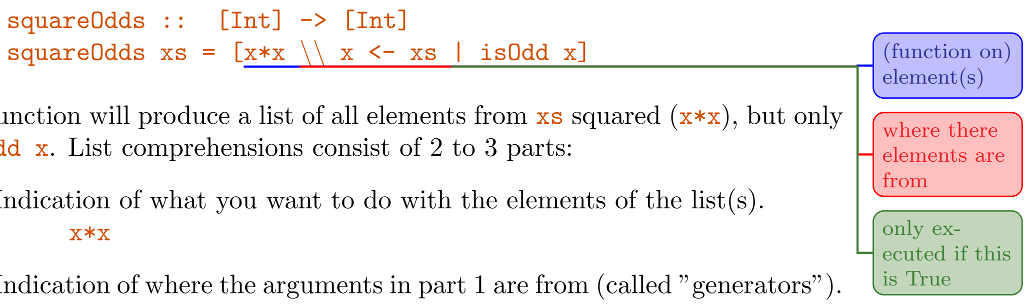
Subsetting a List

Apparently there's functions for lists that you can use to get specific elements or a subset, who'd've thought.

Function/Operator	Meaning
`!!`	returns the n'th element from xs (starts at 0)
`take n xs`	returns the first n amount of elements from xs
`drop n xs`	returns xs without the last n amount of elements
`% xs (x,y)`	returns a "slice" of xs, from index x to y

List Comprehensions

I'm afraid list comprehensions won't help you comprehend lists. They're just another way to format lists: they're a way to quickly apply simple recursive functions to all elements in a list:



```
squareOdds ::  [Int] -> [Int]
squareOdds xs = [x*x \\ x <- xs | isOdd x]
```

This function will produce a list of all elements from `xs` squared (`x*x`), but only if `isOdd x`. List comprehensions consist of 2 to 3 parts:

1. Indication of what you want to do with the elements of the list(s).
   `x*x`
2. Indication of where the arguments in part 1 are from (called "generators").
   `\\ x <- xs`
3. (optional) Makes sure to only include the current element(s) if what comes after the"`|`" is `True`.
   `| isOdd x`

When applying list comprehensions to multiple lists, there's 2 different ways to combine the elements: nested and parallel (i.e. pairwise).

```
[(x,y) \\ x <- [1,2,3], y <- [a,b,c]]
```

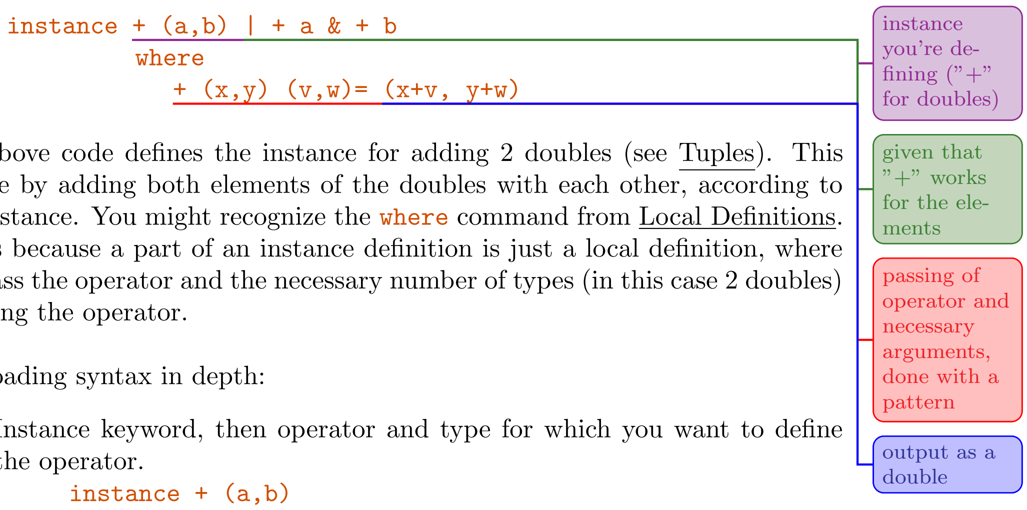
This comprehension makes a list of tuples: `[(Int,Char)]`. And utilizes a ",". Comma's are used for nested generators, meaning that it'll loop through all possible combinations and produce the list
`[(1,a),(1,b),(1,c),(2,a),(2,b),(2,c),(3,a),(3,b),(3,c)]`.

```
[(x,y) \\ x <- [1,2,3] & y <- [a,b,c]]
```

This comprehension makes a list of the same type, but utilizes a `&`. This means that it only executes part one for all elements of the same index, i.e. pairwise combinations. This also means that if one list is longer than the other, the length of the list it produces is the same as that of the smallest list (which also means that the usage of infinite lists is perfectly fine). This comprehension produces the list `[(1,a),(2,b),(3,c)]`

3.4 Overloading

Overloading is the act of defining operators for a new type, whether it's a tuple, record type or algebraic type. But because it's generally done in 1 line (which I'm not gonna do) the structure can be confusing.



```
instance + (a,b) | + a & + b
         where
            + (x,y) (v,w)= (x+v, y+w)
```

instance you're defining ("+" for doubles)
given that "+" works for the elements
passing of operator and necessary arguments, done with a pattern
output as a double

The above code defines the instance for adding 2 doubles (see Tuples). This is done by adding both elements of the doubles with each other, according to this instance. You might recognize the `where` command from Local Definitions. That's because a part of an instance definition is just a local definition, where you pass the operator and the necessary number of types (in this case 2 doubles) for using the operator.

Overloading syntax in depth:

1. Instance keyword, then operator and type for which you want to define the operator.
   `instance + (a,b)`
2. Requirements for being able to execute this operator for this type.
   `| + a & + b`
3. Keyword for local definition.
   `where`
4. Passing of operator and necessary types for executing this operator.
   `+ (x,y) (v,w)`

5. Output of the function.
```
= (x+v, y+w)
```

Overloading a record type

When defining operators for a new record type that contains general types (like "`a`", "`b`", etc.), there's some extra stuff that you have to do:

```
:: BadHombres a b = {Mexico :: a, China :: b}

instance + (BadHombres a b) | + a & + b
         where
            + h1 h2 = {Mexico = h1.Mexico + h2.Mexico,
                       China = h1.China + h2.China}
```

You might've noticed that the definition of `BadHombres` has a different syntax than the example in Record Types. This is because the the requirements for executing the function (`| + a & + b`) need to be able to access the types in the record type.

If you have an already defined type (whether it's a basic type, algebraic type or data structure), you don't need to pass the type. If you always use the same types for `BadHombres`, for example 2 Strings, you can just write:

```
:: BadHombres = {Mexico :: String, China :: String}

instance + BadHombres | + String
         (...)
```

Also note that if you have a general type in your record type, like in `BadHombres a b :: {...}`, you also need to specify the types whenever you make a new `BadHombres`. So like:

```
CrookedHillary :: BadHombres String String
CrookedHillary = {Mexico = "Rapists", China = "TradeWar"}
```

Keep in mind that you don't *need* to define the type of `CrookedHillary`, since `Mexico` and `China` together always form the type `BadHombres`. So just the 2nd line is enough.

4 Functions

In a way, a function is a type as well. Just one that requires one or more arguments (which also means you can pass a function as an argument, cause it's technically a type, but we'll get to that in Higher Order Functions).

Also note that for a function to work, its types don't always need to be defined, but defining types narrows the scope of a function. For example, when you have a function without type definitions that multiplies two arguments, you'd better hope no one is gonna try to pass characters or strings to the function. Therefore it's generally safer (and usually required) to define the types of a function.

Functions in clean have the following basic format:

```
Contains ::  Char String -> Bool | hd String & tl String
Contains c "" = False
Contains c s
            | c == hd s = True
            | otherwise = Contains c tl(s)
```

input and output types

given that functions hd and tl work for String

'if empty string'. you can have separate functions for different inputs

if-statement. Expects boolean

single = assigns value to function

Clean functions can be divided into 5 parts, with part 1, 2 and 4 being optional, depending on the situation:

1. Name and type definition.

   ```
   Contains ::  Char String -> Bool
   ```

2. Requirements for being able to execute the function: "Given that you know how to...". Use `&` if multiple requirements.

   ```
   | hd(String) tl(String)
   ```

3. Cases for certain inputs. (called "patterns")

   ```
   Contains c ""
   Contains c s
   ```

4. Logical Condition(s). (preceded by "guards")

   ```
   | c == hd s
   | otherwise
   ```

5. Assigning a value to the function.

   ```
   = True
   = Contains tl(s)
   ```

Clean's "single line programming" approach makes it so that you often need multiple functions for a single exercise. If you get stuck and don't know how to write the function, see Writing a Function.

4.1 Recursion

Recursion is the act of making a function call itself with different arguments/inputs. In clean, recursion is used a lot, mainly because it does not have loops.

When writing a recursive function, ALWAYS include the termination case. For example: `Contains c "" = False`, then write the code for when the termination case is false. Also make sure that this code will eventually reach the termination case.

4.1.1 For-loop in Clean

(The following is more like an example of a Clean function, rather than actual study material)

Clean for-loops use Local Definitions to iterate a command a certain number of times. The hard part here is to figure out what should be done in the case of `i > j` because it still needs to be executed in one line of code. This is the basic structure though.

```
ForLoop ::  Int (...)  -> a
ForLoop i (...)  = SubLoop i 0 (...)
                   where
                     SubLoop i j (...)
                         | i > j = ...SubLoop i (j+1) (...)
                         | otherwise = (...)
```

input & output may vary

local definition

also varies

4.2 Local Definitions

Clean's functional nature requires you to use a lot of separate functions. Luckily, you can just define them within the function you need them in, though the structure can be confusing:

```
Triangle ::  Int -> String
Triangle n = SubTriangle n 0
      where
      SubTriangle n m
      | n >= 0 = space (n-1) +++ line (1+2*(m-n)) +++ "\n"
+++ SubTriangle (n-1) m
```

actual values passed to SubTriangle

keyword for local def.

definition of SubTriangle with parameters

Note that when defining a local definition, you don't need to specify the types of the parameters.

Local definitions can access any parameter in the function they're in, but not the other way around. Which is similar to the way that brackets work in Java.

4.3 λ - Abstractions

Clean gets scared when it sees multiple lines, so on top of local definitions, it also supports λ-abstractions. Like local definitions, they're functions used locally, only without a name and signified with a \.

```
doubleList ::  [Int] -> [Int]
doubleList xs = map subdouble xs
      where
      subdouble x = x * 2

doubleList ::  [Int] -> [Int]
doubleList xs = map (\x = x * 2) xs
```

Both functions double every element in an Int list (kudos to Matthijs), but the latter uses λ-abstraction. Within the parentheses and after a \, it just behaves as a regular function, with the arguments on the left hand side, and the output on the right hand side. Also, you cannot use guards in a λ-abstraction, nor can it call itself. So you'll have to use The If-Operator and Accumulators.

4.3.1 Accumulators

UPDATE: It's part of the exam after all.

Accumulators are mainly used in λ-abstractions as limited way of making them recursive. This is because a λ-abstraction does not have a name and therefore cannot call itself. It can instead use an accumulator, which stores the previous outcome of the function. Please note that there's only a single slide that covers accumulators and personally I can't make chocolate of it.

```
1.)  sumList xs = foldl (+) 0 xs

2.)  sumList xs = foldl (\x acc = acc + x) 0 xs

3.)  sumList xs = subSum 0 xs
     where
        subSum acc [] = acc
        subSum acc [x:xs] = subSum (acc + x) xs

Start = sumList [1,2,3]
```

(For more info on `foldl` see `foldr` and `foldl`)

The above functions all add every element of a list to each other, but the last 2 use accumulators. However, it looks like function 3 might have just used any other name instead of acc and still work the exact same, so I'm not sure to what extent it's actually an accumulator.

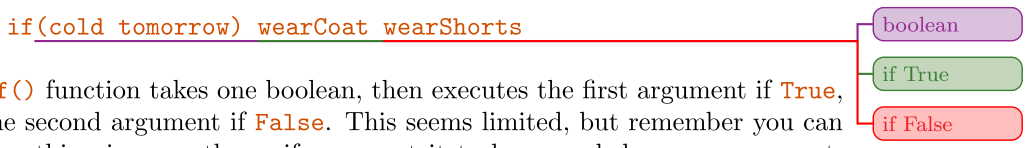
To me, using an accumulator in a λ-abstraction makes the most sense. Since you can only pass it known arguments, the code-word acc must (in my logic) automatically indicate an accumulator. As opposed to function 3 where it can be any other word or letter. Plus, as I've said before, it can help to make up for that fact that a λ-abstraction cannot call itself.

So let's take function 2 as our example: the initial value of an accumulator is always `zero`, which means it can also be used for any other data type that supports that value, not just numbers. Anyways, `foldl` is executed as long as it gets an argument (in this case `x`), so we move to the first element of the list, `1`, and execute the function: `x + acc = 1 + 0 = 1`. This value is then stored in the accumulator, since that was the last output of the λ-abstraction. Next, `foldl` moves on to the 2nd element: `x + acc = 2 + 1 = 3`. This is then stored in the accumulator again, and the next and last element of the list is passed: `x + acc = 3 + 3 = 6`. At this point `foldl` runs out of arguments and outputs the last computed value of the λ-abstraction (or the accumulator, since it stores the last output anyways).

The weird thing between function 1 and 2 is that the accumulator in 2 doesn't seem to even use the `0` that's passed to `foldl`, whereas in function 1 it *is* used. This is more noticeable when using a similar structure to calculate the length of a list. Also, this has probably more to do with `foldl` than with the accumulator, but keep it in mind.

4.4 The If-Operator

You can use Clean's if-operator as an alternative for guards. Which can be helpful in λ-abstractions because you can't use guards there.



```
if(cold tomorrow) wearCoat wearShorts
```

The `if()` function takes one boolean, then executes the first argument if `True`, and the second argument if `False`. This seems limited, but remember you can put something in parentheses if you want it to be regarded as one argument. This includes another if-statement, resulting in an "if, else if" situation.

4.5 Higher Order Functions

Because functions are technically types too, they can also be used as arguments. These kinds of functions are called "higher order functions". Though dissecting higher order functions can be confusing:

```
f1 a b c = a c (b c)
```

This is an example of a higher order function without type definitions. At first glance, it seems like the function outputs two arguments and one tuple that's missing a comma. Which shouldn't be possible. And it isn't: some of these arguments are functions.

To figure out the structure of these functions, you have to look at the right hand side of the `=` sign, from which you can concludee a few things:

1. `a` is a function, since more arguments follow.
2. `c` is not a function, since (within the parentheses) no arguments follow. Also, `c` cannot be a function with `b` and `c` as arguments, since that means

c is a function with 2 arguments, and no arguments at the same time.

3. b is a function, since (within the parentheses) one argument follows.

This means that a is the main function, with type `t t -> t`, since it takes two arguments and returns one. b has type `t -> t` and c has type t. Meaning that the types of function f1 are:

```
f1 ::  (a a -> a) (a -> a) a -> a
f1 a b c = a c (b c)
```

The type of the output is whatever is after the last "->"

As you can see, when defining a type as a function, you need to use parentheses "()". This also holds when you define one function to be another function entirely:

```
f2 ::  a -> a
f2 x = x

f3 ::  (a -> a)
f3 = f2
```

Tips for Structure of Higher Order Functions

Look at the right hand and left hand side of the function and use logic to determine which arguments are functions and which are arguments. Then determine how many arguments each function takes and, if needs be, the specific type of their input and output. Some tips:

- Look at the arguments in the pattern, and define them one by one. If it is a function, use parentheses.
- Look at what arguments need to be put together in order to make a single type.
- When needing to put arguments together, the first one is always a function and anything following (including the stuff in parentheses) are arguments for that function. Ex: `(a b,c d)` is a tuple, which has as type `(t,t)`. Therefore a is a function with b as an argument, etc.
- Something in parentheses is always a function of its own. Ex: `a c (b c)` here a is a function, c and `(b c)` are arguments. And within the parentheses, b is a function with c as an argument.
- If you define one of your types as a function, and it also happens to have another function inside of it (like `a c (b c)`), you don't need to specify the types of that function within the function you're defining, just defining the number of arguments is enough: a as `(t t -> t)`.
- If you wanna be really cheeky, defining the wrong types can cause Clean to print the cases for which the function doesn't work in the error window. Sometimes simply copying the types of one of these cases fixes your problem.

4.5.1 Currying

If only it was as nice as actual curry.

Currying can only be done with Higher Order Functions. It's when you leave out one or more arguments from the function in your higher order function. You can then place that argument, after calling the higher order function itself. For example, the following function takes a function and a number:

```
negate ::  (a -> a) a -> a | ~ a
negate f x = ~f x

square x = x * x

Start = negate square 2
```

This will return `-4`, the negation of the argument squared. However, you can also omit `x` in `negate`, by which you curry the function:

```
curriedNegate ::  (a -> a) -> (a -> a) | ~ a
curriedNegate f = ~f

Start = negate square 2
```

This will also return `-4`, but clean chops the code up differently. First it calls the function `negate`, which only takes one argument as you can see in the type definition. So Clean will initially only look for one argument and read the code as `(negate square) 2`.

The outcome however, is also a function which takes one argument. So clean sees that the code in parentheses has a function as output, and then looks at the next piece of code for the argument for that function, which in this case is `2`.

Keep in mind that you can only curry the last arguments of a function. If you wanna curry an argument but put it somewhere in the middle of your function, you have to use some forbidden Clean magic that you couldn't find even if you tried.

4.6 `foldr` and `foldl`

Never thought these shitty functions would be important enough to be an exam topic.

In order to understand `foldr` and `foldl`, you need to under stand how Clean reads lists. When you make the list `[1,2,3]` Clean interprets it as `[1:[2:[3:[]]]]`.

Both `fold` functions require an operator / function, a starting value and a list, in that order. They then apply that function to the list, much like `map`. But unlike `map`, it combines elements of a list in a certain way, rather than changing its elements.`foldr` as the following structure (might help you understand it):

```
foldr op r [a:x] = op a (foldr op r x)
foldr op r [] = r
```

There's 2 ways to use the `fold` functions, with an operator and with a function (which is often a λ-abstraction).

Let's use `foldr` with `(+)` and `0` on our previous list. The function changes all the `:` into the operator you passed it, and puts a `0` in the empty list to the right (if using `foldl` it'll place it on the left). Meaning that `[1:[2:[3:[]]]]` turns into `(1+(2+(3+(0))))`, which is `6`.

You might be wondering why you even need the `0`. That's because different operators have different base values. When you add `0` to something, it stays the same. If we wanted to get the product of a list instead of the sum, we would write `foldr (*) 1 xs`, because multiplying something by `1` doesn't change that something.

For the other way of using the `fold` functions (with a function instead of an operator) I'm gonna use a λ-abstraction with an accumulator and if-statement, because that's the only way I know how to do it.

The following λ-abstraction basically returns `x` if it's higher than the accumulator, otherwise it returns the accumulator, which stores the previous outcome of the λ-abstraction (see λ - Abstractions, Accumulators, and The If-Operator if you need to know more).

```
foldl (\x acc = if(x > acc) x acc) 0 xs
```

Applying this to `foldr` means that it loops through every element in the list and returns the highest number.

According to my Clean logic, which isn't always right cause it's Clean after all, the only major difference between the operator version and the function version of `foldr` and `foldl` is that when passing it a function it actually doesn't seem to utilize the number you pass it. But I might be wrong about that.

5 Tips, Tricks & Miscellaneous

5.1 Program Reasoning

5.1.1 Induction

Induction can seem really confusing, but it's actually really simple. Just like in formal reasoning, there's a template for proving induction. The only parts that you actually have to think about are the base case and the induction step.

In this example we'll use the operator `++` and the function `map`, which applies a function to every element in a list. Let's say you're given the following definition of the `++` operator on lists, and the `map` function:

```
(++) ::  [a] [a] -> [a]
(++) [] ys = ys                        (1)
(++) [x:xs] ys = [x :  xs ++ ys]       (2)

map ::  (a -> b) [a] -> [b]
map f [] = []                          (3)
map f [x:xs] = [f x :  map f xs]       (4)
```

And you have to prove that `map f (as ++ bs) = (map f as) ++ (map f bs)`. So the mapping of the combined list of `as` and `bs` is the same as the mapping of `as` combined with the mapping of `bs`.

The numbered lines in the definitions of `++` and `map` are free to use in your proof.

```
Prove:  for all as ::  [a] :  map f (as ++ bs) = (map f
as) ++ (map f bs)
Proof:  by induction on as
```

First up is the proof of the base case. For this, just make the argument you're applying induction to `zero`. You always apply induction to only 1 argument, which in this case is `as`, so let's take `as` and make it `[]`. Now you have to transform the equation so that both sides are equal:

```
Base case:
assume as = []

Prove:  map f (as ++ bs) = (map f as) ++ (map f bs)

Proof:
(ass.)        map f (as ++ bs) = (map f as) ++ (map f bs)
(1)       <=> map f ([] ++ bs) = (map f []) ++ (map f bs)
(3)       <=> map f bs = (map f []) ++ (map f bs)
(1)       <=> map f bs = [] ++ (map f bs)
          <=> map f bs = map f bs
```

Note that the "<=>" is required for the proof, as are the indications of what rules you used (like "(3)").

With the base case proven, now we move on to the induction step. Proving the induction step is similar to the base case in the way that you aim to make both sides of the equation equal. But before that, you have to substitute a part of your induction step by your induction hypothesis (IH). The induction step for a list is always the IH with an extra head:

```
Induction Case:
Assume property holds for certain as:
map f (as ++ bs) = (map f as) ++ (map f bs)      (IH)

Prove:map f ([a:as] ++ bs) = (map f [a:as]) ++ (map f bs)
      for all a

Proof:
(4)        map f ([a:as] ++ bs) = (map f [a:as])++(map f bs)
      <=> map f ([a:as] ++ bs) = [f a:map f as] ++ (map f bs)
(IH)  <=> map f ([a:as] ++ bs) = [f a] ++ (map f as) ++ (map f bs)
(2)   <=> map f ([a:as] ++ bs) = [f a] ++ map f (as ++ bs)
      <=> map f [a:as ++ bs]  = [f a] ++ map f (as ++ bs)
(4)   <=> map f [a:as ++ bs]  = [f a :  map f (as ++ bs)]
      <=> [f a :  map f (as ++ bs)] = [f a :  map f (as ++ bs)]

Base + Induction proof complete.
```

And that's the format of induction proof that you have to use.

5.1.2 Applicative vs. Normal Order

A program can be evaluated in 2 ways: via applicative order (arguments first), and via normal order (left-most function first). Both aim to reduce the program / function to its *normal form*, which is the form when nothing needs to be computed anymore, i.e. the output of the function.

As for an example, take the following code:

```
square n = n * n
inc n = n + 1
square_inc n = square (inc n)

Start = square_inc 7
```

This is reduction by **applicative order**, which is what Clean uses:

```
Start
= square_inc 7
= square (inc 7)
= square (7 + 1)
= square 8
= 8 * 8
= 64
```

And this by **normal order**, which is what Java uses.

```
Start
= square_inc 7
= square (inc 7)
= (inc 7) * (inc 7)
= (7 + 1) * (inc 7)
= 8 * (inc 7)
= 8 * (7 + 1)
= 8 * 8
= 64
```

5.2 Programming & Exam Tips

5.2.1 Writing a Function

There's no one way to write a function, but there are some general steps you can take with which you can cheese more than you'd think:

1. First determine the output of the function, and the ways you can (combine stuff to) get that output. e.g. a function that loops through a list and returns a `Bool`, can just call itself with the tail of the list, since that function is also a `Bool`, and if you need to output a list or a list of lists, keep the ways you can combine lists in mind, like `:` or `++`.

2. Divide the stuff that you need to do into chunks, while thinking of Clean's limitations and possibilities (don't thinks of the exact syntax yet). Also check the StdEnv, maybe there's a function in there that already solves a part of your problem.

3. If you need arguments that aren't passed by the original function and that are always the same, or just something that loops a certain number of times, use a local definition. Local definitions can be stacked, so don't skimp on 'em.

4. Write the patterns for the possible cases, starting with the most specific cases (like the base case) and work your way to the most general one.

5. If there's multiple cases that have the same pattern (e.g. when 2 arguments are the same, do this. Otherwise, do that) use guards. You can also stack guards, but make sure the indent is correct.

6. If you're still stuck after this, it's probably because of a trick with a specific type that you need to use (like the difference between the `,` and `&` operators in list comprehensions), or some special Clean magic that only the TA's and Peter know.

In short, the golden combination for writing a function is usually determining the output type(s), patterns, local definitions and guards. As for some general tips when writing a function:

- When writing the actual code, remember that a single `=` sign is always used to assign a value to either the output, or a type in the output (like in a record type).
- Don't forget the StdEnv!
- I can't stress this enough but make sure you know the difference between assigning a type (`::`) and assigning a value (`=`). This is especially important when working with record types.
- If you get a "cannot unify types" error but you haven't got a clue why, try deleting the functions type definition, cause sometimes you just didn't write it properly.

5.2.2 Determining a Function's Output

When trying to figure out the output of a function (something that's going to be asked on the exam), you have to look at the code like Clean does, meaning that you first have to calculate the stuff "in parentheses". For example, in the first exercise of the exam of June 2016 the output of the function `same_begin [3,2,2,1,2] [3,2,1,2]` is asked, with as function body:

```
same_begin as bs = map fst (takeWhile (\(a,b) ->a == b)
                    [ (a,b) \\ a <- as & b <- bs ])
```

Start left, and determine how many and what kind of arguments each part needs. `map` is a function that needs a function and a list. But the list is not that clear, cause it's this long thing in parentheses. So you first change that part into a list.

`takeWhile` is a function of type `(a -> Bool) [a] -> [a]`. And it outputs all elements of a list for which the boolean holds. In this case the boolean is the λ-abstraction `(\(a,b) = a == b)`, which returns `True` is both elements of the double are equal. This boolean is then applied to the list `[(a,b) \\ a <- as & b <- bs]`. Note the `&`, this means that it combines the elements of the list pair wise (so only the ones with the same index).

Now that you know what each part produces, combine them into the solution of the function:

1. `as = [3,2,2,1,2]` and `bs = [3,2,1,2]`. The list comprehension outputs the pairwise combinations. so:
   `[(3,3), (2,2), (2,1), (1,2)]`
2. `takeWhile` only takes the doubles whose elements have the same value:
   `[(3,3), (2,2)]`

3. `fst` takes the first element of a double, and `map` applies it to the elements of the list we just made. Making the eventual output of the function:

    `[3,2]`

One more cheeky trick: if you don't understand the definition of a function in the exam, the name can actually reveal quite a bit.

5.3 Error Message Comprehension

Even reading ancient Greek would be easier than deciphering Clean's error window. Here's something to help you out:

- **Linker error: could not create 'C:\Users...Exercise.exe'**

  Don't worry, you just didn't close your previous .exe window. Close it and try again.

- **Error [Exercise.icl,1,Start]: has not been declared**

  Your code is missing the 'Start =' command.

- **Type error [Exercise.icl,31,Start]:"..." cannot unify types: [Int]**
  Int

  You messed something up with the brackets. (), [] *or* {}

- **Parse error [Exercise.icl,31;4,Type name]: upper case ident expected instead of <type>**

  You need to capitalize the first letter of your new type.

- **Parse error [FQL.icl,58;116,nested guards]: sorry, but for the time being there is a default alternative for nested guards expected**

5.4 Troubleshooting

5.5 Secret StdEnv Commands

Because there's not really a way to Ctrl + F at specific pages, I decided to make the list of stdEnv commands a separate document: https://www.overleaf.com/read/sjdrqxdrfkcd *Won't be working on this until after the exam. Use Functional Programming in Clean for now.*

5.6 Helpful Links

Functional Programming in Clean (2002)
http://www.mbsd.cs.ru.nl/publications/papers/cleanbook/CleanBookI.pdf

A Concise Guide to Clean StdEnv (2011)
http://www.mbsd.cs.ru.nl/publications/papers/2010/CleanStdEnvAPI.pdf