Advanced Programming 2025
Get Familiar With Clean

March 13 2025

1 Goal

The goal of this exercise is to familiarize yourself with the Clean programming language. You are going to create:

- One module, named `Q`, that implements an abstract data type, `Q`, to represent rational numbers ($\mathbb{Q}$). The module is in two files: one implementation (`Q.icl`) and one definition module (`Q.dcl`).
- One main module, named `LetExpr`, that imports `Q`, to represent simplified let-expressions.

While doing so, you will practice with recursive algebraic data types that have infix data constructors, implement instances of overloaded functions and operators, and define a new type constructor class.

During the tutorial you are welcome to experiment with any feature of the Clean programming language.

2 Assignment

2.1 Module Q

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

- The operators `==` and `<` test whether two fractions are equal or smaller respectively.
- The operators `+`, `-`, `*`, and `/` implement the mathematical operations $+$, $-$, $\cdot$, and $\div$. Division by zero is illegal and should be `abort`ed with a suitable error message.
- The constants `zero` and `one` generate the neutral values for addition and multiplication respectively.
- Just like other numbers, fractions are signed numbers (can be negative, positive, or zero). `abs` takes the absolute value, `sign` returns the sign of the argument, and `~` flips the sign.
- The operation `toReal` turns a fractional value into a `Real`. We introduce a new type constructor class, `toQ`, to convert values into a fractional value: for the `Real` instance you obviously need to implement some sort of approximation (keep it simple). `toQ (d,n)` should result in the representation of $\frac{d}{n}$, and `toQ (x,d,n)` should result in the representation of $x\frac{d}{n}$.
- To print fractional numbers, implement an instance of type `Q` for `toString`. If the fractional value $x\frac{d}{n}$ happens to correspond with a whole number ($d = 0$ or d is divided with remainder zero by n), it should be printed as a whole number. Otherwise, it should be delimited by round brackets, and if $x = 0$, then x should not be printed.

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`.

2.2 Module LetExpr

Create the main module `LetExpr` that imports module `Q`. In `LetExpr` you are going to manipulate representations of expressions of the following form:

```
E1 = (let x = 42 - 3 in x / 0) + (let y = 6 in y * y) // syntactic correct expression with run-time error
E2 = let x = 42 in x + (let x = 58 in x)              // syntactic correct expression with result 100
E3 = let x = 1 in let y = 2 in let x = 3 in 4         // syntactic correct expression with result 4
E4 = let x = 1 in x + y                               // syntactic incorrect expression (y not bound by let)
E5 = (let x = 1 in x) * x                             // syntactic incorrect expression (outer x not bound by let)
E6 = let x = x in 42                                  // syntactic incorrect expression (cyclic let definition)
E7 = let x = 4/3 in let y = 3/4 in x * y              // syntactic correct expression with result 1
```

These expressions can be represented by utilizing the following types:

```
:: LetExpr = NUM Q
           | VAR Name
           | (+.) infixr 6 LetExpr LetExpr
           | (-.) infixr 6 LetExpr LetExpr
           | (*.) infixr 7 LetExpr LetExpr
           | (/.) infixr 7 LetExpr LetExpr
           | LET Name LetExpr LetExpr
:: Name :== String
```

As such, an expression can be a fractional value v (represented as `(NUM` v`)`), a variable with the name x (represented as `(VAR` x`)`), an arithmetical operation $+$, $-$, $\cdot$, and $\div$ with the infix data constructors `+.`, `-.`, `*.`, and `/.` (we need to add `.` to the data constructor names to distinguish them from the arithmetical operators), and finally, a let definition (**let** x `=` e_1 **in** e_2) (represented as `(LET` x e_1 e_2`)`). The above expressions E_i are represented by the following `LetExpr` definitions `E` i:

```
E 1 = LET "x" (NUM (toQ 42) -. NUM (toQ 3)) (VAR "x" /. (NUM zero)) +. LET "y" (NUM (toQ 6)) (VAR "y" *. VAR "y")
E 2 = LET "x" (NUM (toQ 42)) (VAR "x" +. LET "x" (NUM (toQ 58)) (VAR "x"))
E 3 = LET "x" (NUM one) (LET "y" (NUM (toQ 2)) (LET "x" (NUM (toQ 3)) (NUM (toQ 4))))
E 4 = LET "x" (NUM one) (VAR "x" +. (VAR "y"))
E 5 = (LET "x" (NUM one) (VAR "x")) *. VAR "x"
E 6 = LET "x" (VAR "x") (NUM (toQ 42))
E 7 = LET "x" (NUM (toQ (4,3))) (LET "y" (NUM (toQ (3,4))) (VAR "x" *. VAR "y"))
```

2.2.1 Printing let-expressions

Make an *instance* of the overloaded function `toString` for `LetExpr`:

```
instance toString LetExpr
  where toString // your implementation is here
```

This instance should display `LetExpr` values as shown above, using the `toString` instance of `Q` values from module `Q`.

The result of `Start = [toString (E i) \\ i <- [1..7]]` is the list with subsequent strings $E_i (1 \leq i \leq 7)$, perhaps differing in the use of brackets and rational numbers.

Tip: module `Text` offers the useful `String` conversion and concatenation operator `<+`.

2.2.2 Free variables

Implement the function `free :: LetExpr -> [Name]` that yields all free variables present in an expression. A variable x is bound by a **let** x `=` $\ldots$ **in** e. A variable is free if it is not bound. The result of `Start = [free (E i) \\ i <- [1..7]]` is `[[], [], [], ["y"], ["x"], [], []]`. Note that in `E 5` the variable with the name `"x"` *is* bound in the first argument of the multiplication, but not in the second argument.

Tip: module `StdList` from `StdEnv` and module `Data.List` contain useful list operations.

2.2.3 Unused variables

In expression `E 3` variables with the names `"x"` and `"y"` are defined, but not used. This expression can be simplified to `(NUM (toQ 4))` without changing its 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 :: LetExpr -> LetExpr` that performs this transformation. The result of `[remove_unused_lets (E i) \\ i <- [1..7]]` is `[E 1, E 2, NUM (toQ 4), E 4, E 5, NUM (toQ 42), E 7]`.

2.2.4 Cycle detection

Implement the function `cyclic :: LetExpr -> Bool` that determines whether the expression contains at least one cyclic let definition. In general a let definition (**let** x `=` e_1 **in** e_2) is cyclic if x occurs free in e_1. The result of `[cyclic (E i) \\ i <- [1..7]]` is `[False, False, False, False, False, True, False]`.

2.2.5 Substitution

A let definition (**let** x `=` e_1 **in** e_2) can be reduced by uniformly substituting every *free* occurrence of x in e_2 by e_1. We denote this with e_2 `<@` (x, e_1). For instance, `(x + (let x = 58 in x)) <@ (x,42)` is changed into `42 + (let x = 58 in x)`, `(let y = 2 in let x = 3 in 4) <@ (x,1)` is changed into **let** `y = 2` **in let** `x = 3` **in** `4`, and `(let y =` $\frac{3}{4}$ **in** `x * y) <@ (x,`$\frac{4}{3}$`)` is changed into **let** `y =` $\frac{3}{4}$ **in** $\frac{4}{3}$ `* y`.

Implement this substitution step with the operator `(<@) :: LetExpr (Name,LetExpr) -> LetExpr`.

2.2.6 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 bound variables, and as such expressions E_4 and E_5 have no value. It is also only possible if an expression is not cyclic, so expression E_6 also has no value. The value of an arithmetical operation is the arithmetical operation on the values of its arguments. Division by zero is not allowed, and as such expression E_1 has no value. The value of a let definition is the use of its new definition in the remainder of the expression, as calculated by the substitution operator `<@`. Note that this means the value of E_2 is 100, and not 84 or 116.

If an expression can not be evaluated for one of the above reasons, evaluation yields `?None`, and otherwise it yields `?Just v`, with `v` the value result of the evaluation.

Implement the function `eval :: LetExpr -> ?Q` calculating the value v of an expression and, if it exists, yielding that value as `(?Just` v`)`. If the expression is not a valid calculation, `?None` should be yielded.

The result of `Start = [toString (eval (E i)) \\ i <- [1..7]]` is `["?None", "(?Just␣100)", "(?Just␣4)", "?None", "?None", "(?Just␣42)", "(?Just␣1)"]`.

Tip: If you use `toString` here, then you need to implement an instance of `?a` for `toString` as well. The correct header of this instance definition is:

```
instance toString (?a) | toString a
  where toString // your implementation is here
```

Deadline

To receive feedback, hand in your solution before April 9 23:59h.