In [75]:
```
import $file.hw4stdlib
import hw4stdlib._
```

```
Compiling hw4stdlib.sc
```

Out[75]:
```
import $file.$

import hw4stdlib._
```

Homework 4

This is a longer assignment due to the exam. Due 10/3 at 11:59pm

Submission Instructions

Upload only this .ipynb file to Canvas. Do not add anything to stdlib since you can't submit it.

In this homework we will define integers and their operations, then build an interpreter.

See [This link (https://www.notion.so/Guidelines-for-Programming-Homework-dbd25efa7bb24915ae6bcb06827fc5b6)](https://www.notion.so/Guidelines-for-Programming-Homework-dbd25efa7bb24915ae6bcb06827fc5b6) for what is and isn't allowed in your code.

Problem 1 (20 pts)

For the arithmetic interpreter we will write later on in the homework we will need arithmetic operations for the integers. These will obviously correspond very closely to the operations defined on the Natural Numbers. But now we will need to account for negative numbers in a proper way.

Start by defining the Algebraic Datatype(sealed trait with case classes) for the Integers. The encoded type should be:

$$\begin{aligned}\mathbb{Z} ::= &\ \text{Positive } \mathbb{N} \\ &\mid \text{Negative } \mathbb{N}\end{aligned}$$

We will use this encoding because it will allow us to use some of our definitions for the Natural Numbers to define the operations on the Integers. This is the core strategy of functional programming: using smaller functions to build up larger ones. This is true even across types. We want to always minimize the ammount of code we have be reusing what we have already constructed, whenever possible.

1a

Define the ADT for Integers below:

In [76]:
```scala
// BEGIN SOLUTION
sealed trait Integer
case class Positive(x : Nat) extends Integer
case class Negative(x : Nat) extends Integer
// END SOLUTION
```

Out[76]:
```
defined trait Integer
defined class Positive
defined class Negative
```

In [77]:
```scala
def ione : Integer = Positive(Succ(Zero))
def ineg_one : Integer = Negative(Succ(Zero))

def int_to_str(x : Integer) : String = x match {
    case Positive(x) => nat_to_str(x)
    case Negative(x) => "-" + nat_to_str(x)
}

def print_integer(x : Integer) = println(int_to_str(x))

print_integer(ione)
print_integer(ineg_one)
passed(3)
```

```
1
-1

*** Tests Passed (3 points) ***
```

Out[77]:
```
defined function ione
defined function ineg_one
defined function int_to_str
defined function print_integer
```

Absolute Value and Negation

Below are defined the absolute value function and a negation function

In [78]:
```scala
def abs(x : Integer) : Nat = x match {
    case Positive(x) => x
    case Negative(x) => x
}

def negate(x : Integer) : Integer = x match {
    case Positive(x) => Negative(x)
    case Negative(x) => Positive(x)
}
```

Out[78]:
```
defined function abs
defined function negate
```

1b: Addition

Define plus and minus for Integers. Don't use the versions we created for ℕ as it did some funky things to get minus to behave correctly. Try and create these from scratch instead. It is recommended to use the lte(less than or equals) function we defined last week along with abs to make the job easier.

Note that we have renamed the operations for natural numbers so that they are of the form:

```
nat_<operation name>
```

For instance, plus has been renamed to `nat_plus` . This was done so we don't have namespace conflicts

In [79]:
```
// BEGIN SOLUTION
def plus(n : Integer, m : Integer) : Integer = (n, m) match {
    case (Positive(x), Positive(y)) => Positive(nat_plus(x, y))
    case (Negative(x), Negative(y)) => Negative(nat_plus(x,y))
    case (Negative(x), Positive(y)) => nat_lte(x, y) match {
        case True  => Positive(nat_minus(y, x))
        case False => Negative(nat_minus(x, y))
    }
    case (Positive(x), Negative(y)) => nat_lte(x, y) match {
        case True  => Negative(nat_minus(y, x))
        case False => Positive(nat_minus(x, y))
    }
}
// END SOLUTION
```

Out[79]: defined function plus

In [80]:
```
assert(plus(Positive(three), Negative(six)) == Negative(three))
assert(plus(Positive(three), Negative(two)) == Positive(one))
assert(plus(Positive(five), Positive(three)) == Positive(eight))
passed(5)
```

```
*** Tests Passed (5 points) ***
```

1c: Subtraction

Implement subtraction below Hint: Subtraction is very easy if you use `plus` and a `negate`

In [81]:
```
// BEGIN SOLUTION
def minus(x : Integer, y : Integer) : Integer = plus(x, negate(y))
// END SOLUTION
```

Out[81]: defined function minus

In [82]:
```
assert(minus(Positive(three), Negative(six)) == Positive(nine))
assert(minus(Positive(three), Negative(two)) == Positive(five))
assert(minus(Positive(five), Positive(six)) == Negative(one))
passed(3)
```

```
*** Tests Passed (3 points) ***
```

1d: Multiplication

Write multiplication for Integers. You should be able to use `nat_mult` to greatly simplify this

In [83]:
```
// BEGIN SOLUTION
def mult(x : Integer, y : Integer) : Integer = (x,y) match {
    case (Positive(x), Positive(y)) => Positive(nat_mult(x, y))
    case (Negative(x), Negative(y)) => Positive(nat_mult(x, y))
    case (Negative(x), Positive(y)) => Negative(nat_mult(x, y))
    case (Positive(x), Negative(y)) => Negative(nat_mult(x, y))
}
// END SOLUTION
```

Out[83]:
```
defined function mult
```

In [84]:
```
assert(mult(Positive(three), Negative(two)) == Negative(six))
assert(mult(Positive(two), Positive(one)) == Positive(two))
assert(mult(Negative(three), Negative(three)) == Positive(nine))
passed(5)
```

```
*** Tests Passed (5 points) ***
```

1e: Exponentiation

Recall that for `pow` we will restrict ourselves to only positive powers. Use the definition of `mult` from above so that your polarity(Positive/Negative) is correct. Recall the cases for $-x^n$ for even vs odd n. A hint for your base case: `pow(x, 0) = Positive( 1 )`

In [85]:
```
// BEGIN SOLUTION
def pow(x : Integer, y : Nat) : Integer = y match {
    case Zero => Positive(Succ(Zero))
    case Succ(y) => mult(x, pow(x, y))
}
// END SOLUTION
```

Out[85]:
```
defined function pow
```

In [86]:
```
assert(pow(Negative(two), two) == Positive(four))
assert(pow(Positive(three), one) == Positive(three))
assert(pow(Negative(two), three) == Negative(eight))
passed(3)
```

```
*** Tests Passed (3 points) ***
```

The Arithmetic Language: Our First Interpreter

Now we are ready to define our first interpreter. We will define the Arithmetic language syntax below as a sealed trait. It will be your job to correctly construct the interpreter for it based on the inference rules we covered in class. Recall that each rule corresponds to a case in the `eval` function.

In [87]:
```
sealed trait Expr
case class Num(x : Integer) extends Expr
case class Plus(x : Expr, y : Expr) extends Expr
case class Minus(x : Expr, y : Expr) extends Expr
case class Mult(x : Expr, y : Expr) extends Expr
case class Pow(x : Expr, y : Nat) extends Expr
```

Out[87]:
```
defined trait Expr
defined class Num
defined class Plus
defined class Minus
defined class Mult
defined class Pow
```

Problem 2 (10 points)

Now that we have defined the syntax for Arithmetic expressions. Go ahead and define the interpreter. We have given the signature for the function. (Bonus points if you define and use the helper function `eval-bin` that we discussed in class.

Recall that the type of this interpreter should be $eval : \mathrm{Expr} \to \mathbb{Z}$

In [88]:
```
def eval_bin(f : ((Integer, Integer) => Integer), e1 : Expr, e2 : Expr)
    f(eval(e1), eval(e2))

def eval(expr : Expr) : Integer = expr match {
    case Num(n) => n
    case Plus(e1, e2) => eval_bin(plus, e1, e2)
    case Minus(e1, e2) => eval_bin(minus, e1, e2)
    case Mult(e1, e2) => eval_bin(mult, e1, e2)
    case Pow(e1, n) => pow(eval(e1), n)
}
```

Out[88]:
```
defined function eval_bin
defined function eval
```

In [89]:
```
val x: Expr = Num(Positive(six))
assert(eval(x) == Positive(six))
passed(4)
```

```
*** Tests Passed (4 points) ***
```

Out[89]:
```
x: Expr = Num(Positive(Succ(Succ(Succ(Succ(Succ(Succ(Zero))))))))
```

In [90]:
```
val x2: Expr = Plus(Num(Positive(two)), Num(Positive(two)))
assert(eval(x2) == Positive(four))
passed(3)
```

```
*** Tests Passed (3 points) ***
```

Out[90]:
```
x2: Expr = Plus(Num(Positive(Succ(Succ(Zero)))),Num(Positive(Succ(Succ
(Zero)))))
```

In [91]:
```
val x3: Expr = Mult(Plus(Num(Positive(two)), Num(Positive(two))), Num(Ne
assert(eval(x3) == Negative(eight))
passed(3)
```

```
*** Tests Passed (3 points) ***
```

Out[91]:
```
x3: Expr = Mult(Plus(Num(Positive(Succ(Succ(Zero)))),Num(Positive(Succ
(Succ(Zero))))),Num(Negative(Succ(Succ(Zero)))))
```

Problem 3 (10 points)

Implement equality for $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, and $\text{List } a$

Most should have the form:

$$eq : A \to A \to \mathbb{B}$$

Where you will want to fill each A with the type of equality you are defining

3a: $\mathbb{B}$

Implement `bool_eq`

In [92]:
```
// BEGIN SOLUTION
def bool_eq(x : Bool, y : Bool) : Bool = (x, y) match {
    case (True, True) => True
    case (False, False) => True
    case _ => False
}
// END SOLUTION
```

Out[92]:
```
defined function bool_eq
```

In [93]:
```
assert(bool_eq(True, True) == True)
assert(bool_eq(False, True) == False)
assert(bool_eq(True, False) == False)
assert(bool_eq(False, False) == True)
passed(2)
```

```
*** Tests Passed (2 points) ***
```

3b: ℕ

Implement `nat_eq`

In [94]:
```
// BEGIN SOLUTION
def nat_eq(x : Nat, y : Nat) : Bool = (x, y) match {
    case (Zero, Zero)           => True
    case (Succ(px), Succ(py)) => nat_eq(px, py)
    case _                      => False
}
// END SOLUTION
```

Out[94]:
```
defined function nat_eq
```

In [95]:
```
assert(nat_eq(ten, ten) == True)
assert(nat_eq(ten, Zero) == False)
assert(nat_eq(five, six) == False)
passed(3)
```

```
*** Tests Passed (3 points) ***
```

3c: ℤ

Implement `int_eq`

In [96]:
```
// BEGIN SOLUTION
def int_eq(x : Integer, y : Integer) : Bool = (x, y) match {
    case (Positive(x), Positive(y))       => nat_eq(x, y)
    case (Negative(x), Negative(y))       => nat_eq(x, y)
    case (Positive(Zero), Negative(Zero)) => True
    case (Negative(Zero), Positive(Zero)) => True
    case _                                => False
}
// END SOLUTION
```

Out[96]:
```
defined function int_eq
```

In [97]:

```
assert(int_eq(Positive(nine), Positive(nine)) == True)
assert(int_eq(Negative(eight), Negative(eight)) == True)
assert(int_eq(Positive(nine), Negative(nine)) == False)
assert(int_eq(Positive(nine), Positive(Zero)) == False)
assert(int_eq(Positive(five), Positive(six)) == False)
passed(3)
```

```
*** Tests Passed (2 points) ***
```

3d: List a

Implement `list_eq`. Since lists are polymorphic, your function needs to take a third parameter which should be the `eq` function for the given a:

$$eq : \text{List } A \to \text{List } A \to (A \to A \to \mathbb{B}) \to \mathbb{B}$$

In [98]:

```
// BEGIN SOLUTION
def list_eq[A](l1: List[A], l2: List[A], eq: (A, A) => Bool): Bool =
    (l1, l2) match {
        case (Empty, Empty) => True
        case (Cons(x1, t1), Cons(x2, t2)) => and(eq(x1, x2), list_eq(t1,
        case _ => False
    }
// END SOLUTION
```

Out[98]:

```
defined function list_eq
```

In [99]:

```
assert(list_eq(Empty, Empty, nat_eq) == True)
assert(list_eq(Empty, Empty, bool_eq) == True)
assert(list_eq(Cons(True, Empty), Cons(True, Empty), bool_eq) == True)
assert(list_eq(Cons(True, Empty), Empty, bool_eq) == False)
assert(list_eq(Cons(True, Cons(False, Empty)), Cons(True, Cons(True, Emp
passed(4)
```

```
*** Tests Passed (4 points) ***
```