Assignment 1

James Zhang zhany111

October 5, 2019

1 Task 1

1)

1. neg (exp (neg (5) abs (6)))
   in math: $-((-5)^{|6|})$
2. plus (plus (plus (5 (6 (7 8)))))
   in math: $5 + 6 + 7 + 8$
3. exp (plus (times (17 24) 14) 2)
   in math: $(17 * 24 + 14)^2$

2 Task2

1)

1. $plus(a, b) := int * int \rightarrow int$ output $:= a + b$
2. $minus(a, b) := int * int \rightarrow int$ output $:= a - b$
3. $times(a, b) := int * int \rightarrow int$ output $:= a * b$
4. $power(a, b) := int * int \rightarrow int$ output $:= a^b$
5. $neg(a) := int \rightarrow int$ output $:= -a$
6. $abs(a) := int \rightarrow int$ output $:= |a|$

2)

1. $neg(a) := int \rightarrow int$ output $:= -a$
2. $abs(a) := int \rightarrow int$ output $:= |a|$

3 Task 3

The following part is computed in Ruby:

Part 1

Design a calculator by defining class IExpr. IExpr contains number that can be int or float. We can calculate between different IExpr variables using methods defined in IExpr class.

Every method in IExpr returns a new IExpr class variable.

```
# coding: utf-8
class IExpr
  # Task 1
  attr_accessor :number

  # constructor of IExpr. We don't need to worry about types
  # that much because int can be calculated with float in ruby
  def const(n1)
    # Raise error if the input is not an integer
    if not(n1.is_a?(Integer))
      raise "can only calculate int"
    else
      @number = n1
      return self
    end
  end

  #Easily define methods to calculate numbers.
  #In each method, return value is self
  def neg(n1)
    @number = -n1.number
    return self
  end

  def abs(n1)
    #check if input >= 0 or not
    if n1.number >= 0
      @number = n1.number
    else
```

```
      @number = 0-n1.number
    end
    return self
  end

  def plus(n1, n2)
    @number = n1.number + n2.number
    return self
  end

  def times(n1, n2)
    @number = n1.number * n2.number
    return self
  end

  def minus(n1, n2)
    @number = n1.number - n2.number
    return self
  end

  def exp(n1, n2)
    @number = n1.number ** n2.number
    return self
  end

  # the function that return the value of IExpr
  def interpret
    return @number
  end

end
```

Part 2 We then design a new class called IBExpr that can calculate on bools. It is similar to IExpr in part 1, except we calculate bools

```
# coding: utf-8
$LOAD_PATH << '.'
require 'IExpr.rb'
# Easily construct IBExpr class that oprates on bool types
class IBExpr
  #Task 2
  attr_accessor :b
  # In each method, return self
  def ttt()
    @b = true
    return self
  end

  def fff()
    @b = false
    return self
  end

  def lnot(a)
    @b = not(a.b)
    return self
  end

  def land(a, c)
    @b = a.b && c.b
    return self
  end

  def lor(a, c)
    @b = a.b || c.b
    return self
  end

  #Output the bool value of IBExpr
  def interpret
    return @b
```

```
    end
end
```

4 Task 4

The following code is computed in Fsharp

Part 1

Design IExpr class along with the function that calculate IExpr classes. We assume IExpr can only contain integer value.

```
module IExpr
// Define IExpr type that can contain only int number
type IExpr() = class
    let mutable number = 0
    // method that return the value of IExpr
    member this.Num() =
        number
    // Method that construct a new IExpr with input int
    member this.Const(a : int) =
        number <- a
        this
end

// Define functions that calculate on IExpr types.
// For all the functions, the inputs are IExpr types
// The output is a new IExpr.
// They are easy to understand
let plus(a : IExpr, b : IExpr) =
    let number = a.Num() + b.Num()
    let c = IExpr()
    c.Const(number)

let minus(a : IExpr, b : IExpr) =
    let number = a.Num() - b.Num()
    let c = IExpr()
    c.Const(number)

let times(a : IExpr, b : IExpr) =
    let number = a.Num() * b.Num()
    let c = IExpr()
    c.Const(number)
```

```
let rec exp(a : IExpr, b : IExpr) =
    // We don't want negative power because IExpr only
    // can calculate on int
    if b.Num() < 0 then
        failwith "no negative power"
    else
        let mutable n = a.Num()
        // Calculate using for loop
        for i in 0 .. b.Num() do
            n <- n * a.Num()
        IExpr().Const(n)

let abs(a : IExpr) =
    let n = a.Num()
    let mutable number = 0
    if n > -n then
        (number <- n)
    else (number <- -n)
    let c = IExpr()
    c.Const(number)

let neg(a : IExpr) =
    let number = -a.Num()
    let c = IExpr()
    c.Const(number)
// Function that output the int that input IExpr contains.
let interpret(a : IExpr) =
    a.Num()
```

Part 2

Most of the code is same as in the Part 1. But IExpr can contain string now! We assume that we cannot do any calculation on IExpr with string contained. So fail when calculate on IExpr with string.

```
module IExpr_with_subst
// The IExpr class that can contain strings, can also substitube number
// into strings.
// Most of the codes are same as in Part 1, just a few changes.
type IExpr() = class
    let mutable number = 0
    // with a initial value of letter, able to check if the IExpr.letter is e
    let mutable letter = ""

    member this.Num() =
       // check if this IExpr contains a letter, if yes then fail because
       // This method is only called on calculating numbers but not on letter
        if letter <> "" then failwith "cannot calculate letters"
        else
            number
    // Method that return the letter IExpr contains.
    member this.Letter() =
        letter
    // method that create a IExpr contains string
    member this.Ident(s : string) =
        letter <- s
        this

    member this.Const(a : int) =
        number <- a
        this
end

// Function that change s to a if s = a.
// We assume a is substituded only when s = b.letter()
// Same as in Part 1. All function returns a new IExpr.
let subst(s : string, a : IExpr, b : IExpr) =
    // fail when we connot subsitude anything
    if s = "" then failwith ("nothing to substitube")
```

```
    elif b.Letter() = "" then failwith ("nothing to be substituded")
    else
        // As we assumed, substitude only when s = b
        if b.Letter() = s then
            let c = a.Num()
            IExpr().Const(c)
        else
            b
// Function that output the number IExpr contains
let interpret(a : IExpr) =
    // Check if IExpr contains a letter or not, if yes
    // fail because we connot interpret letters
    if a.Letter() <> "" then
        failwith "cannot interpret expresion with letters"
    else
        a.Num()

// Codes below are the same as in Part 1
let rec exp(a : IExpr, b : IExpr) =
    if b.Num() < 0 then
        failwith "no negative power"
    else
        let mutable n = a.Num()
        for i in 0 .. b.Num() do
            n <- n * a.Num()
        IExpr().Const(n)
let plus(a : IExpr, b : IExpr) =
    let number = a.Num() + b.Num()
    let c = IExpr()
    c.Const(number)

let minus(a : IExpr, b : IExpr) =
    let number = a.Num() - b.Num()
    let c = IExpr()
    c.Const(number)

let times(a : IExpr, b : IExpr) =
    let number = a.Num() * b.Num()
    let c = IExpr()
```

```
    c.Const(number)


let abs(a : IExpr) =
    let n = a.Num()
    let mutable number = 0
    if n > -n then
        (number <- n)
    else (number <- -n)
    let c = IExpr()
    c.Const(number)

let neg(a : IExpr) =
    let number = -a.Num()
    let c = IExpr()
    c.Const(number)
```