Assignment 4: Exercises in programming with the *supersub* language

1 Assignment Overview

This is the last project of the programming languages class. This class is specifically for implementation using the *supersub* language that you created as a result of the assignment number 3. This assignment will be graded differently than the previous programming projects. In this project there are eight 10 point problems giving a total of 80 points, six 20 point problems giving a total of 120 points, and four 25 point problems giving an additional 100 points. Therefore there is a total of 300 points worth of problems.

For undergraduates the grading will follow the following rule, for a student that completes 100 or less points of this project, the point total will be the number of points earned. Therefore, if a student only successfully completes the 10 point problems, then the student will receive an 80 on this project. However, if a student completes more than 100 points worth of projects then the student can earn up to 20 extra bonus points. For each 10 points earned over 100 points is worth one bonus point. Thus if an undergraduate student completes all 300 points, then the student will receive a $100 + 0.1 * 200 = 120$ grade (100 percent plus 20 point bonus).

For graduate students the requirement is a bit different. In order to earn 100 points the graduate student will need to complete 200 points of exercises, so as long as the graduate student completes less than 200 points of exercises, then his score will be half of the completed points. The graduate student can also earn a grade of up to 120 because each 10 points earned over 200 will count for 2 points instead of one, thus a graduate student that earns 300 hundred points will receive a $0.5 * 200 + 0.2 * 100 = 120$ grade (100 percent plus 20 point bonus).

1.1 Evaluation

Each problem will be submitted in a separate file in the `supersub` language. The files will be named for the submitted problem in the following fashion: Problem 10.1 will be submitted in the file `P101.in`, while problem 20.3 will be submitted in the file `P203.in` and so on. These files will define the combinators needed to implement the combinator(s) described in the problem description. The problems will be run on a reference implementation of supersub. This reference implementation will expand all combinator definitions and use a canonical bound variable naming to ensure all a valid consistent comparisons to reference data. This test will be run by concatenating test files to the end of your provided input file, and the output will be compared against reference solutions. Since the comparison version of the program will expand combinators, do not worry if your output does not include the unexpanded combinators, while the examples provided in this text do. The comparison will be performed on versions of the output that do not include any combinators.

1.2 Submission

This project and any other incomplete projects must be submitted by 11:50pm on April 14, 2017. There will be no late points for this project. Submit as many working problems as you can by this date. If there are any previous projects that have not been submitted by this date, then the student will not receive a passing grade in the class. Make sure any past projects are submitted by this time. I will send a reminder message to students that have incomplete project submissions a week before this deadline.

Submit the last project to `mycourses` as a zip or tar file of the project solution files named according to the conventions listed in the table:

filename	Problem
`P101.in`	Problem 10.1
`P102.in`	Problem 10.2
`P103.in`	Problem 10.3
`P104.in`	Problem 10.4
`P105.in`	Problem 10.5
`P106.in`	Problem 10.6
`P107.in`	Problem 10.7
`P108.in`	Problem 10.8
`P201.in`	Problem 20.1
`P202.in`	Problem 20.2
`P203.in`	Problem 10.3
`P204.in`	Problem 10.4
`P205.in`	Problem 10.5
`P206.in`	Problem 20.6
`P251.in`	Problem 25.1
`P252.in`	Problem 25.2
`P253.in`	Problem 25.3
`P254.in`	Problem 15.4

Each file should be a valid *supersub* program that defines the combinators defined in the problem. The files can include the material that is provided in the file `given.in` that is provided with this project. The provided files should not perform any evaluations, these files will be concatenated with evaluation tests that will then be run in a reference *supersub* implementation to check to see if they function properly. If they function properly you will receive full credit for that problem. Problems that do not pass the test will be reviewed to determine if partial credit is warranted (for example, if it passes some but not all of the tests). You do not need to submit this project to the grader.

2 Problems

2.1 Ten Point Problems

Problem 10.1

Write a combinator that takes two arguments that are Church numerals and computes the subtraction of the second argument from the first. Call this combinator `subtract`. Thus an example of its operation is given as follows:

```
evaluate $subtract $Five $Two ;
Expression Evaluates To: {\x.{\y.(x (x (x y)))}}
```

Problem 10.2

Write a combinator called `square` that takes a Church numeral as an argument and returns and returns the square of the number. An example of the proper functioning of this combinator is:

```
evaluate $square $Three ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x y)))))))))}}
```

Problem 10.3

Create a combinator called `even` that takes a Church numeral as an argument and returns true if it is even and false otherwise. The function of this combinator would be as follows:

```
  evaluate $even $Zero ;
  Expression Evaluates To: $True
  evaluate $even $One ;
  Expression Evaluates To: $False
  evaluate $even $Two ;
  Expression Evaluates To: $True
  evaluate $even $Five ;
  Expression Evaluates To: $False
```

Problem 10.4

Write a combinator called `leq` that processes two arguments and returns true if the first argument is less than or equal to the second. An example of this combinator in action is:

```
evaluate $leq $Zero $Zero ;
Expression Evaluates To: $True
evaluate $leq $Five $Three ;
Expression Evaluates To: $False
evaluate $leq $Five $Five ;
Expression Evaluates To: $True
evaluate $leq $Three $Five ;
Expression Evaluates To: $True
```

Problem 10.5

Create the logical function `implies` that implements the following truth table

X	Y	X IMPLIES Y
False	False	True
False	True	True
True	False	False
True	True	True

The correct implementation of this function will provide the following output:

```
evaluate $implies $False $False ;
Expression Evaluates To: $True
evaluate $implies $False $True ;
Expression Evaluates To: $True
evaluate $implies $True $False ;
Expression Evaluates To: $False
evaluate $implies $True $True ;
Expression Evaluates To: $True
```

Problem 10.6

Create the logical function `equiv` that implements the equivalence function that is shown the following truth table

X	Y	X EQUIV Y
False	False	True
False	True	False
True	False	False
True	True	True

The correct implementation of this function will provide the following output:

```
evaluate $equiv $False $False ;
Expression Evaluates To: $True
evaluate $equiv $False $True ;
Expression Evaluates To: $False
evaluate $equiv $True $False ;
Expression Evaluates To: $False
evaluate $equiv $True $True ;
Expression Evaluates To: $True
```

Problem 10.7

Create a combinator that takes one input of a Church numeral and returns two raised to the input's power, e.g create a function that implements $f(i) = 2^i$. This combinator will be called `power2` and its correct operation is demonstrated by:

```
evaluate $power2 $Zero ;
Expression Evaluates To: $One
evaluate $power2 $One ;
Expression Evaluates To: {\x.{\y.(x (x y))}}
evaluate $power2 $Two ;
Expression Evaluates To: {\x.{\y.(x (x (x (x y))))}}
evaluate $power2 $Three ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x y))))))))}}
```

Problem 10.8

When a pair of numbers is needed to be passed to a function of one argument in supersub, you can do this by creating the following combinator

```
combinator makePair \x.\y.\f. ((f x) y) ;
```

You can then access the first second entry using the following combinators:

```
combinator first \f.f (\x.\y.x) ;
combinator second \f.f (\x.\y.y) ;
```

What these combinators do is pass a function into the pair that either copies the first argument and ignores the second or copies the second argument while ignoring the first. Create a set of combinators to create triples. That is, create a combinator `makeTriple` that takes three arguments and returns a triple, and then create the three combinators `getFirst`, `getSecond`, and `getThird` that can access each of the individual components of the triple similar to the `first` and `second` arguments from above.

```
evaluate $makeTriple $One $Two $Three ;
Expression Evaluates To: {\x.(((x $One) $Two) $Three)}
combinator evaluate triple $makeTriple $One $Two $Three ;
evaluate $getFirst $triple ;
Expression Evaluates To: $One
evaluate $getSecond $triple ;
Expression Evaluates To: $Two
evaluate $getThird $triple ;
Expression Evaluates To: $Three
```

2.2 Twenty Point Problems

Problem 20.1

We have covered representing numbers in lambda calculus using a unary representation with Church numerals, but why not construct binary numbers. For example, we could construct the following three argument combinator where the first argument represents the one, the second argument represents a zero, and the third argument represents a terminating function. The number would represent binary numbers in the following fashion:

Binary Number Combinator	Represented Binary Number	Numerical Value
`\o.\z.\f. o f`	1	1
`\o.\z.\f. z (o f)`	10	2
`\o.\z.\f. o (o f)`	11	3
`\o.\z.\f. z (z (o f))`	100	4
`\o.\z.\f. o (z (o f))`	101	5
`\o.\z.\f. o (o (o (z (o f))))`	10111	23

For this problem you need to write a combinator called `convertb2n` that can convert a binary number defined in the above form into a Church numeral. For an example of how this would work consider:

```
evaluate $convertb2n {\o.\z.\f. z f} ;             #binary zero
Expression Evaluates To: $Zero
evaluate $convertb2n {\o.\z.\f. o f} ;             #binary one
Expression Evaluates To: {\x.{\y.(x y)}}
evaluate $convertb2n {\o.\z.\f. z (o f)} ;         #binary two
Expression Evaluates To: {\x.{\y.(x (x y))}}
evaluate $convertb2n {\o.\z.\f. o (z (o f))} ;     # binary five
Expression Evaluates To: {\x.{\y.(x (x (x (x (x y)))))}}
evaluate $convertb2n {\o.\z.\f. z (o (z (o f)))} ; # binary ten
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x (x y))))))))))}}
```

Problem 20.2

Use the factorial function as a template to create a combinator that computes the sum of the first n squares where n is the argument to the combinator. This combinator, called

$$(\$\text{sumsq n}) = \sum_{i=1}^{n} i^2$$

For an example of how this would work consider:

```
evaluate $sumsq $Zero ;
Expression Evaluates To: $Zero
evaluate $sumsq $One ;
Expression Evaluates To: {\x.{\y.(x y)}}
evaluate $sumsq $Two ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x y)))))}}
evaluate $sumsq $Three ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x (x (x (x (x (x y))))))))))))))}}
```

Problem 20.3

Lists can be represented in lambda calculus in similar form to Church numerals, for example a list of three items can be represented in the form $\lambda f.\lambda t.(f\ a_1)((f\ a_2)((f\ a_3)\ t))$ where a_1 is the first element of the list, a_2 is the second element and so forth. In this case the empty list is given by $\lambda f.\lambda t.t$. To push an element on the front of the list we can use a process similar to the successor function. For example consider the following definitions:

```
combinator empty {\f.\t. t} ;
combinator push {\a.\l.(\f.\t. (f a) (l f t))} ;
```

The push combinator works in this way: It creates a f applied to a for the top list entry, and then applies this to the list with f and t substituted for the new list's f and t. To access the top element we can just substitute the head for the true function and to fill in the remainder of the list we can use the identity function. It is not important as true will select the first item. Thus we can define `top` as

```
combinator top {\l. l $True {\x.x}}
```

For this problem you are expected to write the list `concat`, `isempty`, and `listlen` functions. The `concat` function will take two lists and join them to form a combined list with the contents of the first list preceding the contents of the second list. The `listlen` function would return the length of the list as a Church numeral. Finally, the `isempty` function will return true for an empty list and false otherwise. For an example of how this would work consider the example:

```
combinator list1 $push $One ($push $Two ($push $Three $empty)) ;
combinator list2 $push $Four ($push $Five $empty) ;
evaluate $concat $list1 $list2 ;
Expression Evaluates To: {\x.{\y.((x $One) ((x $Two) ((x $Three) ((x $Four) ((x $Five) y)))))}}
evaluate $concat $list2 $list1 ;
Expression Evaluates To: {\x.{\y.((x $Four) ((x $Five) ((x $One) ((x $Two) ((x $Three) y)))))}}
evaluate $concat $empty $empty ;
Expression Evaluates To: {\x.{\y.y}}
evaluate $concat $list1 $empty ;
Expression Evaluates To: {\x.{\y.((x $One) ((x $Two) ((x $Three) y)))}}
evaluate $concat $empty $list2 ;
Expression Evaluates To: {\x.{\y.((x $Four) ((x $Five) y))}}
evaluate $isempty $empty ;
Expression Evaluates To: $True
evaluate $isempty $list1 ;
Expression Evaluates To: $False
evaluate $isempty $list2 ;
Expression Evaluates To: $False
evaluate $listlen $list1 ;
Expression Evaluates To: {\x.{\y.(x (x (x y)))}}
evaluate $listlen $list2 ;
Expression Evaluates To: {\x.{\y.(x (x y))}}
evaluate $listlen ($concat $list1 $list2) ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x y)))))}}
evaluate $listlen $empty ;
Expression Evaluates To: $Zero
```

Problem 20.4

Create a function called `range` that takes two arguments where the first is a low number, n_l, the second is the high number n_h and it returns a list of numbers starting with n_l and incrementing until it include n_h.

```
evaluate $range $Zero $One ;
Expression Evaluates To: {\x.{\y.((x {\z.{\a.a}}) ((x $One) y))}}
evaluate $range $One $One ;
Expression Evaluates To: {\x.{\y.((x $One) y)}}
evaluate $range $Two $Four ;
Expression Evaluates To: {\x.{\y.((x {\z.{\a.(z (z a))}}) ((x {\b.{\c.(b (b (b c)))}})
((x $Four) y)))}}
```

Problem 20.5

Create a combinator that will return the tail of the list, that is a combinator that will implement the `pop` function that will return the list with the first top element removed. At this point note that the lists described are very similar to Church numerals, and that the operation of removing the top element off of the list is very similar to the predecessor function. The implementation of the `pop` combinator will follow a similar architecture.

```
combinator list1 $push $One ($push $Two ($push $Three $empty)) ;
evaluate $top $list1 ;
Expression Evaluates To: $One
evaluate $pop $list1 ;
Expression Evaluates To: {\x.{\y.((x $Two) ((x $Three) y))}}
evaluate $pop ($pop $list1) ;
Expression Evaluates To: {\x.{\y.((x $Three) y)}}
evaluate $pop ($pop ($pop $list1)) ;
Expression Evaluates To: $empty
```

Problem 20.6

In functional programming one encounters common idioms that help in solving common programming challenges. Generally one creates a family of operators to help solve problems in functional languages, each problem is relatively simple, but the composition yields much more powerful structures. For example, in processing lists it is common to implement the `map` operator which takes a function and a list and returns a new list with the function applied to every item in the list. Another common operator is the `apply` operator which takes three arguments, an operator, an identity, and a list and uses the operator to combine the items of the list. For example, summing the contents of a list would be achieved through

```
evaluate $apply $Add $Zero ($range $One $Four) ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x (x y))))))))))}}
```

This apply is summing the numbers from one to four. The first argument is the add operator, the second is zero which is the identity of the add operator, and the third argument generates a list of values from one to four. For this problem implement the map and apply operator and then use it to implement the `sumsq` operator of problem 20.2, call this revised combinator `sumsq2`. The completed implementation of this problem will allow for the following interactions:

```
combinator testlist $push $Three ($push $One ($push $Zero $empty)) ;
evaluate $map $square $testlist ;
Expression Evaluates To:
{\x.{\y.((x {\z.{\a.(z (z (z (z (z (z (z (z (z a)))))))))}})
((x {\b.{\c.(b c)}}) ((x {\d.{\e.e}}) y)))}}
evaluate $apply $Add $Zero $testlist ;
Expression Evaluates To: {\x.{\y.(x (x (x (x y))))}}
evaluate $sumsq2 $Three ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x (x (x (x (x (x y))))))))))))))}}
```

2.3 Twenty Five Point Problems

Problem 25.1

Write a combinator that computes Ackermann's function. The Ackermann function A is a recursive function of two variables defined as,

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

```
evaluate $ackermann $Zero $Zero ;
Expression Evaluates To: {\x.{\y.(x y)}}
evaluate $ackermann $One $One ;
Expression Evaluates To: {\x.{\y.(x (x (x y)))}}
evaluate $ackermann $Two $Two ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x y)))))))}}
evaluate $ackermann $Three $Zero ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x y)))))}}
evaluate $ackermann $Two $Three ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x y)))))))))}}
evaluate $ackermann $Two $Four ;
Expression Evaluates To: {\x.{\y.(x (x (x (x (x (x (x (x (x (x (x y)))))))))))}}
```

Problem 25.2

Write a combinator called `modulo` that computes the modulo operator (remainder operator). This will return the remainder when dividing the first argument by the second argument. A correct implementation of this will be able to perform the following operations:

```
set maxEvalSteps 100000 ;
evaluate $modulo ($square $Five) $Three ;
Expression Evaluates To: {\x.{\y.(x y)}}
evaluate $modulo ($square $Five) $Four ;
Expression Evaluates To: {\x.{\y.(x y)}}
evaluate $modulo ($square $Five) $Five ;
Expression Evaluates To: {\x.{\y.y}}
evaluate $modulo ($square $Three) $Five ;
Expression Evaluates To: {\x.{\y.(x (x (x (x y))))}}
```

Problem 25.3

The Church numerals are simple and closely related to the concept of iteration. They have a problem in that the predecessor function requires iterating through the entire number thus costing $O(n)$ beta reductions to evaluate. The Scott encodings have the advantage that both the successor and predecessor functions can be evaluated with $O(1)$ beta reductions. In the Scott encoding, the zero is represented by $\lambda x.\lambda y.x$ and the successor is $\lambda p.\lambda x.\lambda f fp$, while the predecessor is $\lambda p.p(\lambda x.x)(\lambda x.x)$. These are implemented using:

```
combinator szero {\x.\y.x} ;
combinator succ {\p.\x.\f. f p} ;
combinator pred {\p.p (\x.x) (\x.x) } ;
evaluate $succ $szero ;
Expression Evaluates To: {\x.{\y.(y $szero)}}
evaluate $succ ($succ $szero) ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a $szero)}})}}
evaluate $succ ($succ ($succ $szero) ) ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a {\b.{\c.(c $szero)}})}})}}
evaluate $pred ($succ ($succ ($succ $szero) ) ) ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a $szero)}})}}
```

For this problem you are to implement the `sZtest`, `sAdd` `sMul` and `sSubtract` primitives for the Scott encoded numbers. In addition you are to write a `sconvert` combinator that will convert a church numeral into a Scott encoded number. For examples of the Scott encoding consider the following:

```
combinator evaluate sone $succ $szero ;
combinator evaluate stwo $succ ($succ $szero) ;
evaluate $sZtest $szero ;
Expression Evaluates To: $True
evaluate $sZtest $sone ;
Expression Evaluates To: $False
evaluate $sZtest $stwo ;
Expression Evaluates To: $False
evaluate $sAdd $stwo $sthree ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a {\b.{\c.(c $stwo)}})}})}}
evaluate $sMul $stwo $sthree ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a {\b.{\c.(c {\d.{\e.(e {\f.{\g.(g {\h.{\i.(i
$szero)}})}})}})}})}})}}
evaluate $sSubtract $sthree $sone ;
Expression Evaluates To: {\x.{\y.(y {\z.{\a.(a $szero)}})}}
```

Problem 25.4

Write a combinator that will convert a Church encoded number into the binary encoded number described in problem *20.1*. This combinator will be called `convertn2b`. This conversion is best built with supporting combinators. For example a combinator that implements division by two is helpful. The list facilities developed earlier can also be useful. Take this one in steps. Once you have a working combinator, the conversion should work like the following example:

```
set maxEvalSteps 10000000 ;
evaluate $convertn2b $Zero ;
Expression Evaluates To: {\x.{\y.{\z.z}}}
evaluate $convertn2b $One ;
Expression Evaluates To: {\x.{\y.{\z.(x z)}}}
evaluate $convertn2b $Five ;
Expression Evaluates To: {\x.{\y.{\z.(x (y (x z)))}}}
evaluate $convertn2b $Ten ;
Expression Evaluates To: {\x.{\y.{\z.(y (x (y (x z))))}}}
evaluate $convertn2b ($square $Five) ;
Expression Evaluates To: {\x.{\y.{\z.(x (y (y (x (x z)))))}}}
```