Queens College of CUNY
Department of Computer Science
Programming Languages
(CSCI 316)
Winter 2026

Assignment #11
"Functional Programming Languages"
Due: January 21, 2026

Introduction:

"Functional Programming" is a paradigm that treats computation as the evaluation of mathematical functions, avoiding imperative instruction, mutable state, and side effects. This approach leads to code that is often more predictable and better suited for concurrent and parallel programming.

Functional languages can be categorized as "pure" - strictly enforce functional principles - like Haskell, or impure/multi-paradigm - support functional programming but also allow imperative styles - like Python. Early versions of Lisp were more purely functional, while later versions introduced other features to make it more multi-paradigm.

In this assignment, we experiment with exercises in both pure and impure functional programming languages

Submissions:

In the Google form, please submit:

- Assignmen11.py etc. (source code for Python and the other functional programming languages)
- Assignment11.txt (console output from your program executions)

Tasks:

[1] Create a new Python program Assignment11.py with this constant:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[2] Define a traditional method that find the sum of squares that are even:

```
total = 0
for i in numbers:
     sq = i * i
     if sq % 2 == 0:
        total += sq
print(total)
```

[2] Define another method that uses list comprehension:

```
total = sum([i**2 for i in numbers if i**2 % 2 == 0])
print(total)
```

[3] Define another method that uses more pure functional style:

```
from functools import reduce
# square each number
squares = map(lambda x: x * x, numbers)
```

```
# keep only even squares
even_squares = filter(lambda x: x % 2 == 0, squares)
# sum the result
result = reduce(lambda a, b: a + b, even_squares)
print(result)
```

[4] Combine into one statement

```
from functools import reduce
result = reduce(lambda a, b: a + b, filter(lambda x: x % 2 == 0, map(lambda x: x * x,
numbers)))
print(result)
```

[5] Rewrite in Haskell (v. 1)

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result  = foldl1 (+) (filter even (map (\x -> x * x) numbers))
main :: IO ()
main = print result
```

[6] Rewrite in Haskell (v. 2 - "point free")

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result  = sum . filter even . map (^2) $ numbers
main :: IO ()
main = print result
```

[6] Rewrite in Scheme

```
(define numbers '(1 2 3 4 5 6 7 8 9 10))
(define result
  (apply +
         (filter even?
                 (map (lambda (x) (* x x)) numbers))))
(display result)   ;
```

[7] Rewrite in Lisp

```
(defparameter numbers '(1 2 3 4 5 6 7 8 9 10))
(defparameter result
  (reduce #'+
          (remove-if-not #'evenp
                         (mapcar (lambda (x) (* x x)) numbers))))
(print result)   ;
```

[8] Rewrite in F#

```
let numbers = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let result =
    numbers
    |> List.map (fun x -> x * x)
    |> List.filter (fun x -> x % 2 = 0)
    |> List.sum

printfn "%d" result
```

==========

[9] Try this Haskell example to compute Fibonacci numbers

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
first10 :: [Integer]
first10 = take 10 fibs
main :: IO ()
main = print first10
```

[10] Haskell example to compute the Sieve of Eratosthenes

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [ x | x <- xs, x `mod` p /= 0 ]

main = print (take 10 primes)
```

[11] Scheme example to compute quadratic formula

```
(define (quadratic a b c)
  (let ((discriminant (- (* b b) (* 4 a c))))
    (if (< discriminant 0)
        '()
        (list (/ (+ (- b) (sqrt discriminant)) (* 2 a))
              (/ (- (- b) (sqrt discriminant)) (* 2 a))))))

(quadratic 1 -3 2)
```