Lazy and Strict Exam Solutions

1 2019 Q3

1.1 Part (b)

1.1.1 Code

```
from n = n : from (n+1)
take 0 xs = []
take n (x:xs) x : take (n-1) xs
```

1.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "from" and it will never stop since "from" will generate an infinite list.

```
take 2 (from 42)
take 2 (42 : from (42+1))
take 2 (42 : from (43))
take 2 (42 : 43 : from (43+1))
take 2 (42 : 43 : from (44))
take 2 (42 : 43 : 44 : from (44+1))
take 2 (42 : 43 : 44 : from (45))
...
```

1.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "from" and see that "take" now has enough arguments. "take" will execute then it won't have enough arguments, then executes "from" again. "from" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
take 2 (from 42)
take 2 (42 : from (42+1))
42 : (take (2-1) (from(42+1)))
42 : (take 1 (from(42+1)))
42 : (take 1 ((42+1) : from((42+1)+1)))
42 : (42+1) : (take (1-1) (from(((42+1)+1)+1)))
42 : (42+1) : (take 0 (from(((42+1)+1)+1)))
42 : (42+1) : []
-- We are done computing the list (Evaluatable)
```

2 2018 Q3

2.1 Part (b)

2.1.1 Code

```
quadup n = n : quadup (n+4)
take 0 xs = []
take n (x:xs) x : take (n-1) xs
```

2.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "quadup" and it will never stop since "quadup" will generate an infinite list.

```
take 2 (quadup 10)
take 2 (10 : quadup(10+4))
take 2 (10 : quadup(14))
take 2 (10 : 14 : quadup(14+4))
take 2 (10 : 14 : quadup(14+4))
take 2 (10 : 14 : quadup(18))
take 2 (10 : 14 : 18 : quadup(18+4))
take 2 (10 : 14 : 18 : quadup(22))
...
```

2.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "quadup" and see that "take" now has enough arguments. "take" will execute then it won't have enough arguments, then it executes "quadup" again. "quadup" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
take 2 (quadup 10)
take 2 (10 : quadup (10+4))
10 : (take (2-1) (quadup (10+4)))
10 : (take 1 (quadup (10+4)))
10 : (take 1 : ((10+4) : quadup ((10+4)+4))
10 : (10+4) : (take (1-1) (quadup ((10+4)+4)))
10 : (10+4) : (take 0 (quadup ((10+4)+4)))
10 : (10+4) : []
-- We are done computing the list (Evaluatable)
```

3 2018 Q3

3.1 Part (b)

3.1.1 Code

```
evenup n = n : evenup (n+2)
take 0 xs = []
take n (x:xs) x : take (n-1) xs
```

3.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "evenup" and it will never stop since "evenup" will generate an infinite list.

```
take 2 (evenup 2)
take 2 (2 : evenup(2+2))
take 2 (2 : evenup(4))
take 2 (2 : 4 : evenup(4+2))
take 2 (2 : 4 : evenup(6))
take 2 (2 : 4 : 6 : evenup(6+2))
take 2 (2 : 4 : 6 : evenup(8))
...
```

3.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "evenup" and see that "take" now has enough arguments. "take" will execute then it won't have enough arguments, then it executes "evenup" again. "evenup" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
take 2 (evenup 2)
take 2 (2 : evenup(2+2))
2 : (take (2-1) (evenup(2+2)))
2 : (take 1 (evenup(2+2)))
2 : (take 1 ((2+2) : evenup(2+2))
2 : (2+2) : (take (1-1) (evenup(2+2)))
2 : (2+2) : (take 0 (evenup(2+2)))
2 : (2+2) : []
-- We are done computing the list (Evaluatable)
```

4 2015 Q4

4.1 Part (b)

4.1.1 Code

```
zig n = n : zag (n-1)
zag n = n : zig (n-1)
take 0 xs = []
take n (x:xs) x : take (n-1) xs
```

4.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "zig" and "zag" it will never stop since "zig" and "zag" will generate an infinite list. "zig" and "zag" do not have a base case, in strict languages it will never end even when going down to 0 and negative numbers

```
take 2 (zig 20)
take 2 (20 : zag (20-1))
take 2 (20 : zag (19))
take 2 (20 : 19 : zig (19-1))
take 2 (20 : 19 : zig (18))
take 2 (20 : 19 : 18 : zag (18-1))
take 2 (20 : 19 : 18 : zag (17))
...
take 2 (20 : ... : 2 : 1 : zag (1-1))
take 2 (20 : ... : 2 : 1 : zag 0)
take 2 (20 : ... : 2 : 1 : 0 : zig (0-1))
take 2 (20 : ... : 2 : 1 : 0 : zig (-1))
take 2 (20 : ... : 2 : 1 : 0 : (-1) : zag (-1-1))
take 2 (20 : ... : 2 : 1 : 0 : (-1) : zag (-2))
...
```

4.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "zig" and "zag" and see that "take" now has enough arguments. "take" will execute then it won't have enough arguments, then it executes "zig" and "zag" again. "zig" and "zag" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
take 2 (zig 20)
take 2 (20 : zag (20-1))
20 : (take (2-1) (zag(20-1)))
20 : (take 1 (zag(20-1)))
20 : (take 1 ((20-1) : zig(20-1))
20 : (20-1) : (take (1-1) (zig(20-1)))
20 : (20-1) : (take 0 (zig(20-1)))
20 : (20-1) : []
-- We are done computing the list (Evaluatable)
```

4.2 Part (c)

4.2.1 Part (i)

```
take 0 []
```

4.2.2 Part (ii)

- Impossible, strict languages cannot evaluate more than what a lazy language can

4.2.3 Part (iii)

```
take 2 $ zig 20
```

4.2.4 Part (iv)

```
zig 20
```

5 2014 Q4

5.1 Part (b)

- Answer to paper is in 2018 Q3 (b)

5.2 Part (c)

5.2.1 Part (i)

```
evenup 2
```

5.2.2 Part (ii)

- Impossible, strict languages cannot evaluate more than what a lazy language can

5.2.3 Part (iii)

```
take 2 $ evenup 2
```

5.2.4 Part (iv)

```
take 0 []
```

6 2013 Q4

6.1 Part (b)

6.1.1 Code

```
down n = n : down (n-1)
take 0 xs = []
take n (x:xs) x : take (n-1) xs
```

6.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "down" and it will never stop since "down" will generate an infinite list. "down" will keep decrementing even past 0 and onto the negative numbers. There is no base case causing it to stop executing so a Strict language will continue executing down

```
take 2 (down 42)
take 2 (42 : down (42-1))
take 2 (42 : down (41))
take 2 (42 : 41 : down (41-1))
take 2 (42 : 41 : down (40))
take 2 (42 : 41 : 40 : down (40-1))
take 2 (42 : 41 : 40 : down (39))
...
take 2 (42 : ... : 2 : 1 : down (1-1))
take 2 (42 : ... : 2 : 1 : down 0)
take 2 (42 : ... : 2 : 1 : 0 : down (0-1))
take 2 (42 : ... : 2 : 1 : 0 : down (-1))
take 2 (42 : ... : 2 : 1 : 0 : (-1) : down (-1-1))
take 2 (42 : ... : 2 : 1 : 0 : (-1) : down (-2))
...
```

6.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "down" and see that "take" now has enough arguments. "take" will execute then it won't have enough arguments, then it executes "down" again. "down" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
take 2 (down 42)
take 2 (42 : down (42-1))
42 : (take (2-1) (down(42-1)))
42 : (take 1 (down(42-1)))
42 : (take 1 ((42-1) : down((42-1)-1))
42 : (42-1) : (take (1-1) (down((42-1)-1)))
42 : (42-1) : (take 0 (down((42-1)-1)))
42 : (42-1) : []
-- We are done computing the list (Evaluatable)
```

6.2 Part (c)

6.2.1 Part (i)

```
take 0 []
```

6.2.2 Part (ii)

- Impossible, strict languages cannot evaluate more than what a lazy language can

6.2.3 Part (iii)

```
take 2 $ down 2
```

6.2.4 Part (iv)

```
down 42
```

7 2012 Q1

7.1 Part (d)

7.1.1 Code

```
up n = n : up (n+1)
get 0 xs = []
get 1 xs = []
get n [] = []
get n (x:xs) = x : get (n-2) xs
```

7.1.2 Strict Evaluation

- Strictness is forced to evaluate all expressions, it will start evaluating "up" and it will never stop since "up" will generate an infinite list.

```
take 2 (up 42)
take 2 (42 : up (42+1))
take 2 (42 : up (43))
take 2 (42 : 43 : up (43+1))
take 2 (42 : 43 : up (44))
take 2 (42 : 43 : 44 : up (44+1))
take 2 (42 : 43 : 44 : up (45))
...
```

7.1.3 Lazy Evaluation

- Laziness work on trying to delay computation as much as possible It will compute "up" and see that "get" now has enough arguments. "get" will execute then it won't have enough arguments, then it executes "up" again. "up" won't actually compute the result. It will pass it in memory which is called a thunk. Laziness in general uses more heap memory because of this. But the advantage is that it can deal with inifinite lists.

```
get 4 (up 42)
get 4 (42 : up (42+))
42 : (get (4-2) (up(42+1)))
42 : (get 2 (up(42+1)))
42 : (get 2 ((42+1) : up((42+1)+1))
42 : (42+1) : (get (2-2) (up((42+1)+1)))
42 : (42+1) : (get 0 (up((42+1)+1)))
42 : (42+1) : []
-- We are done computing the list (Evaluatable)
```

7.2 Part (e)

7.2.1 Part (i)

```
get 0 []
```

7.2.2 Part (ii)

- Impossible, strict languages cannot evaluate more than what a lazy language can

7.2.3 Part (iii)

```
get 4 $ up 42
```

7.2.4 Part (iv)

```
up 42
```