

Opgavebesvarelse 8g

Shatin Nguyen (hlv332), Peter Asp Hansen (glt832) & Oliver Fontaine Raaschou (vns328)

23. februar 2023

1 Opgave 8g0

Delopgave a

I denne opgave, vil vi implementere en cyclic queue i F#. Kravene til en cyclic queue er, at vi skal opretholde en **first** og en **last**, hvor **first** er det forreste element i køen og **last** er det bagerste element i køen. Et element i vores kø skal enten være **Some e**, hvor e er en integer eller **None**, hvis elementet er tomt. Vi vil i vores besvarelse, beskrive de forskellige funktioner og tanker omkring implementeringen.

Til at lave programmet, brugte vi 8g_handout.zip, hvor funktionerne allerede var skrevet op samt typen **Value** som en integer. For at holde styr på det både det forreste og bagerste element, lavede vi to **mutable** variabler, da de skal ændre værdi når programmet køres. Det er variablerne **first** som peger til det første element og **last** som peger på det sidste element. Vi lavede også en **mutable queue**, som skal præsentere køen.

```
1 let mutable first : Value Option = None //Peger på det første element
2 let mutable last : Value Option = None //Peger på det sidste element
3 let mutable queue : Value Option [] = [[]] //Præsentere køen
```

Figur 1: Anvendte mutable variabler

Funktion **create** tager en n integer og returnere en **queue** af længden n , som en **unit**. For at lave funktionen bruger vi en **if**-statement, i tilfælde af man sætter $n \leq 0$, returnere funktionen en tom **unit**, i stedet en lang F# fejlbeskrivelse. Når $n > 0$, anvender vi **Array.create** som tager en integer n og en værdi $'T$ og returnere et array med længden n , hvor alle elementerne er $'T$.¹ Vores n skal være værdien som **create** tager imod og vores $'T$ skal være **None**, da det præsentere en tom plads. Vi tildeler operationen til **queue**, så den præsentere vores kø. Som overvejelse, kunne man tildele en position til **first** og **last** i **create**, men vores program fungerede med uden, så vi fandt det ikke nødvendigt at rette.

```
1 let create (n: int) : unit =
2     if n <= 0 then
3         ()
4     else
5         queue <- Array.create n None
```

Figur 2: Funktionen create

¹<https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-arraymodule.html#create>

Funktionen `isEmpty` skal returnere `True`, hvis køen er tom ellers `False`. Vores ide til at bestemme om køen er tom, er ved at sige, hvis den forreste plads er tom, er køen også tom. Vi laver derfor en `if`-statement og siger, hvis `first.IsNone`, så `True` og ellers `False`.

```
1 let isEmpty () : bool =  
2   if first.IsNone then  
3     true  
4   else  
5     false
```

Figur 3: Funktionen `isEmpty`

Funktionen `length` skal returnere længden af køen med elementer, hvilket også er det samme som antal elementer i køen. For at lave funktionen, laver vi to variabler. Den første er `a` som finder længden af hele køen og `b` som finder længden antal `None` i køen. Tager vi så `a-b`, har vi fundet antal elementer i køen, hvilket også er længden af køen.

```
1 let length () : int =  
2   let a = queue.Length  
3   let b = queue |> Array.filter (fun i -> i = None) |> Array.length  
4   let c = a-b  
5   c
```

Figur 4: Funktionen `length`

Funktion `enqueue` skal tage vores et element `e` og returner `True`, hvis det lykkedes at tilføjet et element forrest i køen eller `False`, hvis det ikke lykkedes. For at lave `enqueue`, så opstiller vi nogle krav til funktionen. Kravende er således:

- Hvis køen er tom, så skal `first` og `last` pege på `index[0]` i `queue` og indsætte elementet her, og returnere `True`.
- Hvis køen er fuld, kan man ikke enqueue og der returneres `False`.
- For hvert element tilføjet, skal `last` rykke sig en plads større i køen, altså `indeks[x + 1]`, hvor `x` er dens nuværende plads.
- Hvis `last` peger på den sidste plads i køen, skal `last` pege på `indeks[0]` plads og så tilføje elementet og returnere `True`.

For at lave funktionen, anvender vi `if`-statement således, `if first.IsNone` tjekker om køen er tom, hvis ja, ændres værdierne ifølge det første punkt. `elif (queue.Length = length())`, tjekker om køen er fuld, ved at sige, hvis længden af køen er ligeså stort som antal elementer, så er køen tom. Hvis dette er `True` returneres blot `False`. Hvis ingen af de to statements er rigtige, betyder det at køen indeholder elementer og har ledige pladser. Når dette er sandt bliver vi dog nød til tjekke om `last` er ved slutningen af køen. Hvis dette passer, sætter vi den til at være i starten af køen, tilføjer elementet og returnere `True`. Når vi ned til vores sidste condition og skal til at indsætte vores element, incrementer vi `last` plads med `+1` og tilføjer elementet `e` i den nye `last.value` indeks.

```

1 let enqueue (e: Value) : bool =
2     if first.IsNone then
3         first <- Some 0
4         last <- Some 0
5         queue.[0] <- Some e
6         true
7     elif (queue.Length = length()) then
8         false
9     else
10        if (last.Value = queue.Length-1) then
11            last <- Some (0)
12            queue.[last.Value] <- Some e
13            true
14        else
15            last <- Some (last.Value + 1)
16            queue.[last.Value] <- Some e
17            true

```

Figur 5: Funktionen enqueue

Funktionen dequeue skal returnere det forreste element eller hvis køen er tom så `None`. Ligesom med `enqueue`, så har `dequeue` også operationer den skal udføre. De er:

- Hvis køen er tom, gør intet og returner `None`.
- Hvis køen ikke er tom, tjek om `first` peger på enden af array. Hvis ja, så print værdien som `first` peger på, derefter ændre dens værdi til `None` og så få `first` til at pege indeks[0] i `queue`.

For at lave funktionen, bruger vi igen et `if`-statement og hvis `first.IsNone` er `True`, så er køen tom og funktionen returnere blot `None`. Hvis ikke, så tjek om `first` peger på det sidste plads i array eller ej, og udfør ovenstående operationer i punkt to. Vi har fortolket opgavebeskrivelsen på `dequeue` til at sætte elementet til `None`, i stedet for at beholde den, da man alligevel ikke kan bruge det igen.

```

1 let dequeue () : Value option =
2     if first.IsNone then
3         None
4     else
5         if (first.Value = queue.Length-1) then
6             printfn "%A" queue.[first.Value]
7             queue.[first.Value] <- None
8             first <- Some (0)
9             None
10        else
11            queue.[first.Value] <- None
12            first <- Some (first.Value + 1)
13            None

```

Figur 6: Funktionen dequeue

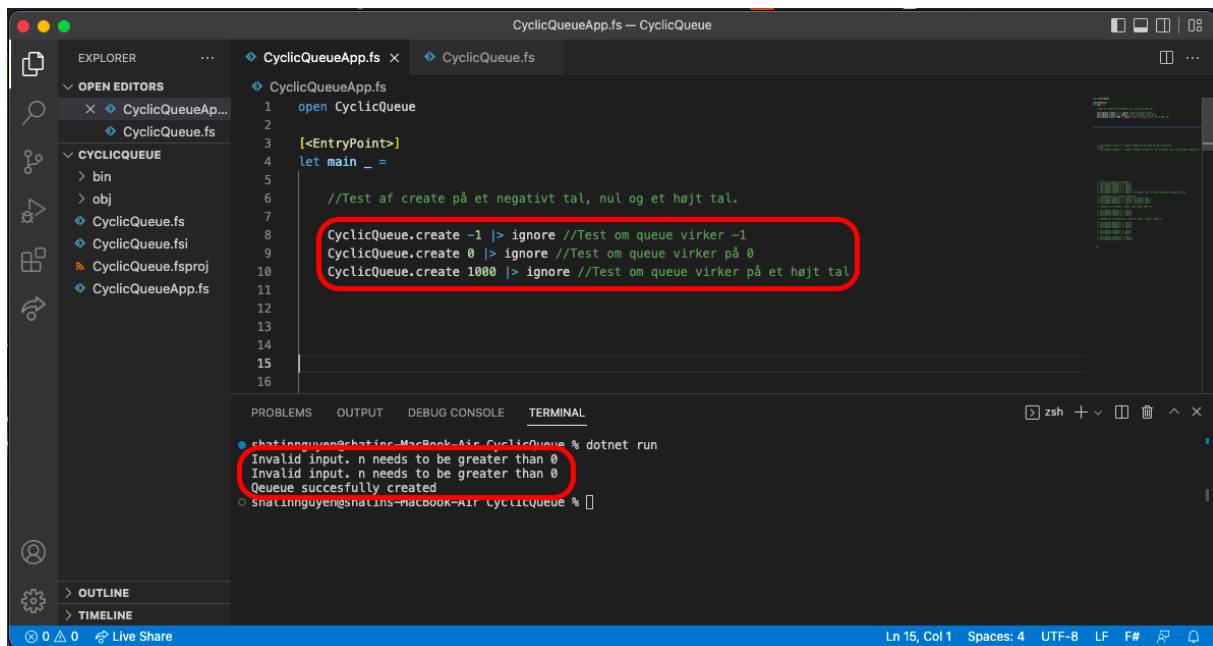
Funktionen `toString` skal konverterer vores elementer i køen til en samlet string. For at lave funktionen, laver vi et array: `arrayValues` med alle værdierne som ikke er `None` med `Array.choose id`². Derefter laver vi en mutable tom string, kaldet `newString` som vi kan ligge elementerne over i. Dette gør vi med et `for`-loop som itererer over 0 til længden af vores nye array. Vi sætter så hvert element i vores nye liste over i `newString`. Til sidst returneres den nye string med værdierne i.

```
1 let toString () : string =
2     let arrayValues = queue |> Array.choose id
3     let lengthOfValues = arrayValues |> Array.length
4
5     let mutable newString = ""
6     for i = 0 to (lengthOfValues-1) do
7         newString <- newString + (string(arrayValues[i])) + " "
8     b
```

Figur 7: Funktionen `toString`

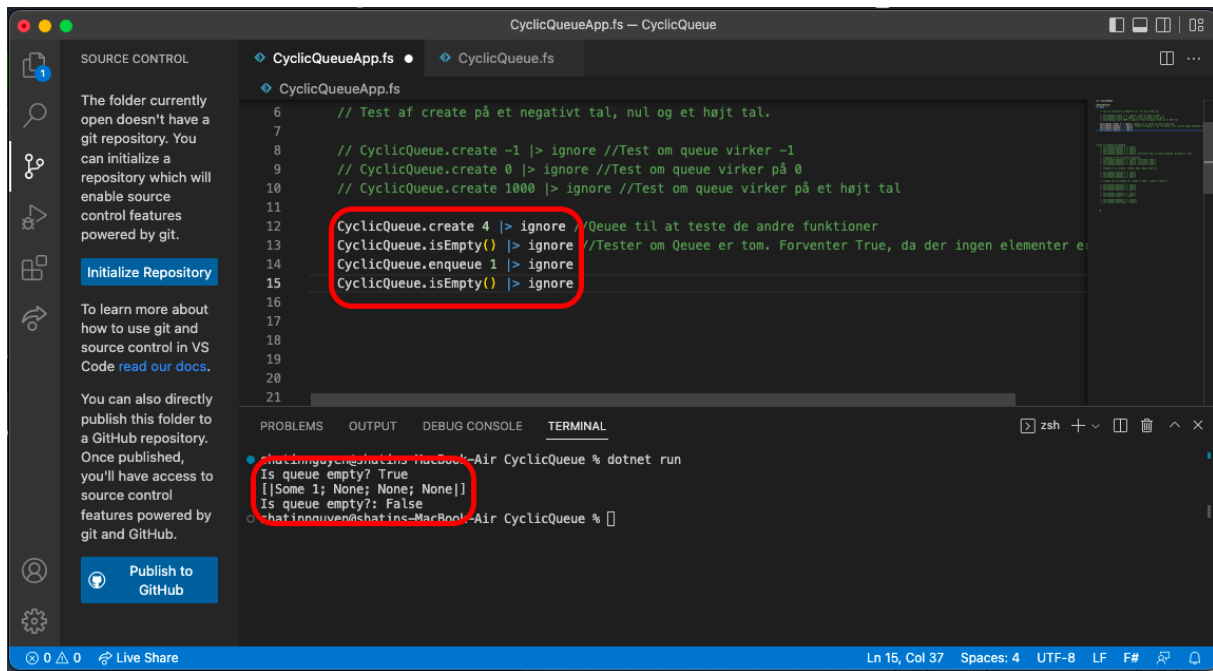
Delopgave b

For at teste programmet, har vi kaldt funktionerne i `CyclicQueueApp.fs`. Vi gør opmærksom på det ikke er alle `print`-statements som dukker op når koden køres. De er anvendt til at teste programmet, men der stadig bevaret i `CyclicQueue.fs`. men er udkommenteret af `//`. Den første funktion vi vil teste er `create`. Her vil vi se, hvad der sker, hvis $n \leq 0$ eller n er et højt tal. I figuren herunder, kan man se at implementeringen lykkedes.



²<https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-arraymodule.html#length>

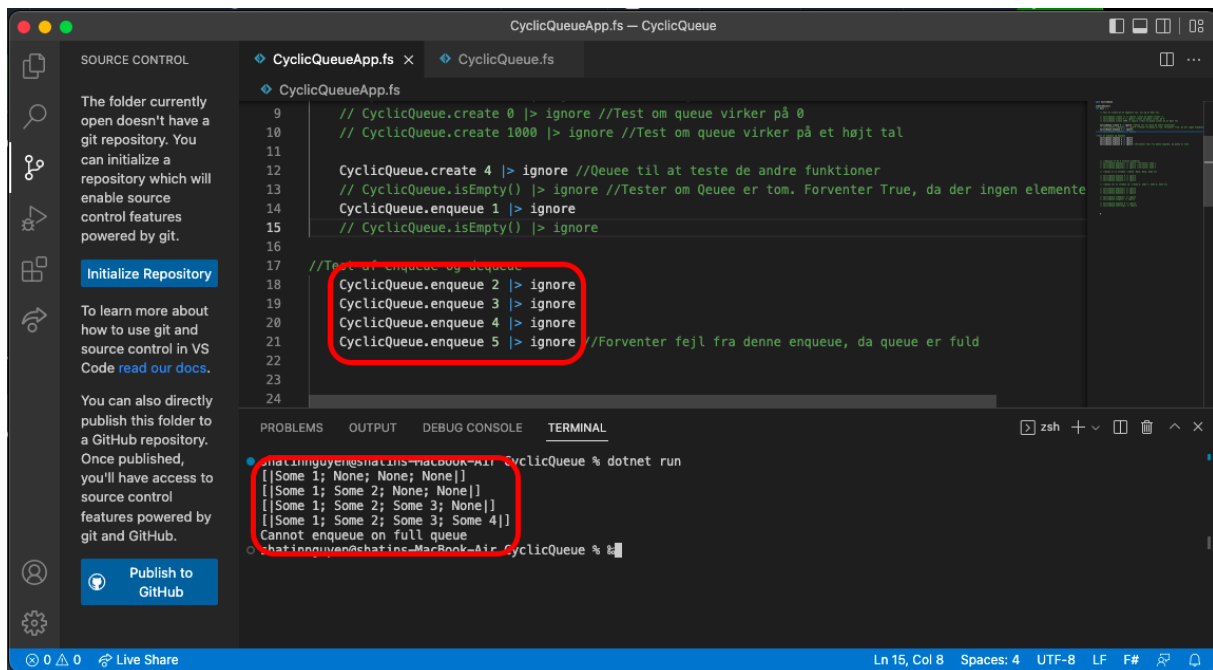
I næste test, vil vi både teste `isEmpty` og `enqueue` på en tom kø. For ikke at gøre testen tung at kører, laver vi en `queue` med fire tomme elementer. Vi kører først `isEmpty` på den tomme `queue`, så `enqueue 1` og så `isEmpty` igen. Med disse operationer, forventer vi `True`, `[|Some 1; None; None; None|]` og `False`. Figuren herunder viser, at vi opnåede de forventede resultater af funktionerne kør fint indtil videre.



```
6 // Test af create på et negativt tal, nul og et højt tal.
7
8 // CyclicQueue.create -1 |> ignore //Test om queue virker -1
9 // CyclicQueue.create 0 |> ignore //Test om queue virker på 0
10 // CyclicQueue.create 1000 |> ignore //Test om queue virker på et højt tal
11
12 CyclicQueue.create 4 |> ignore //Queue til at teste de andre funktioner
13 CyclicQueue.isEmpty() |> ignore //Tester om Queue er tom. Forventer True, da der ingen elementer er
14 CyclicQueue.enqueue 1 |> ignore
15 CyclicQueue.isEmpty() |> ignore
```

```
chatinn@chatins-MacBook-Air CyclicQueue % dotnet run
Is queue empty? True
[|Some 1; None; None; None|]
Is queue empty?: False
chatinn@chatins-MacBook-Air CyclicQueue %
```

Nu kan vi teste at `enqueue` stopper, hvis køen er fuld. Det gør vi blot ved at tilføje fire nye elementer til køen, hvor vi forventer en stopbetingelse ved femte element tilføjet. Figuren herunder viser at alle elementerne tilføjes i rigtige rækkefølger og funktionen stopper når `queue` er fuld.



```
9 // CyclicQueue.create 0 |> ignore //Test om queue virker på 0
10 // CyclicQueue.create 1000 |> ignore //Test om queue virker på et højt tal
11
12 CyclicQueue.create 4 |> ignore //Queue til at teste de andre funktioner
13 CyclicQueue.isEmpty() |> ignore //Tester om Queue er tom. Forventer True, da der ingen elementer
14 CyclicQueue.enqueue 1 |> ignore
15 // CyclicQueue.isEmpty() |> ignore
16
17 //Test af enqueue og dequeue
18 CyclicQueue.enqueue 2 |> ignore
19 CyclicQueue.enqueue 3 |> ignore
20 CyclicQueue.enqueue 4 |> ignore
21 CyclicQueue.enqueue 5 |> ignore //Forventer fejl fra denne enqueue, da queue er fuld
```

```
chatinn@chatins-MacBook-Air CyclicQueue % dotnet run
[|Some 1; None; None; None|]
[|Some 1; Some 2; None; None|]
[|Some 1; Some 2; Some 3; None|]
[|Some 1; Some 2; Some 3; Some 4|]
Cannot enqueue on full queue
chatinn@chatins-MacBook-Air CyclicQueue %
```

Nu kan vi teste vores `dequeue`, hvor vi forventer at det forreste element fjernes fra køen og returneres alene samt at `enqueue`, kan "starter" forfra når `last` peger på det sidste indeks i array. Nedenstående figur viser at både `dequeue` og `enqueue` køres som forventet.

```
CyclicQueueApp.fs
18 CyclicQueue.enqueue 2 > ignore
19 CyclicQueue.enqueue 3 > ignore
20 CyclicQueue.enqueue 4 > ignore
21 CyclicQueue.enqueue 5 > ignore //Forventer fejl fra denne enqueue, da queue er fuld
22
23 // Dequeue af de 3 foreste elementer.
24 CyclicQueue.dequeue() > ignore //Forventer Some 1
25 CyclicQueue.dequeue() > ignore //Forventer Some 2
26 CyclicQueue.dequeue() > ignore //Forventer Some 3
27
28 CyclicQueue.enqueue 6 > ignore
29 CyclicQueue.enqueue 7 > ignore
30 CyclicQueue.enqueue 8 > ignore
31
32 // Queue ser nu således ud: [|Some 6; Some 7; Some 8; Some 4|]
33
34 // CyclicQueue.dequeue() > ignore
35 // CyclicQueue.dequeue() > ignore
36 // CyclicQueue.dequeue() > ignore
```

```
Some 1
[|None; Some 2; Some 3; Some 4|]
Some 2
[|None; None; Some 3; Some 4|]
Some 3
[|None; None; None; Some 4|]
Some 4
[|Some 6; None; None; Some 4|]
Some 5
[|Some 6; Some 7; None; Some 4|]
Some 6
[|Some 6; Some 7; Some 8; Some 4|]
```

Til sidst vil de teste, at `dequeue` begynder forfra når den ankommer til enden af arrayet, samt `length` og `toString` returnere værdierne korrekt.

```
CyclicQueueApp.fs
31 CyclicQueue.dequeue() > ignore
32 CyclicQueue.dequeue() > ignore
33 CyclicQueue.length() > ignore
34 CyclicQueue.toString() > ignore
35
36 CyclicQueue.enqueue 6 > ignore
37 CyclicQueue.enqueue 7 > ignore
38 CyclicQueue.enqueue 8 > ignore
39 CyclicQueue.enqueue 9 > ignore
```

```
Some 4
[|Some 6; Some 7; Some 8; None|]
Some 5
[|None; Some 7; Some 8; None|]
Some 6
[|None; Some 7; Some 8; None|]
Some 7
[|None; Some 7; Some 8; None|]
Some 8
[|None; Some 7; Some 8; None|]
Some 9
[|None; Some 7; Some 8; None|]
```

Vores funktioner giver os de forventede output og de består derfor vores test.

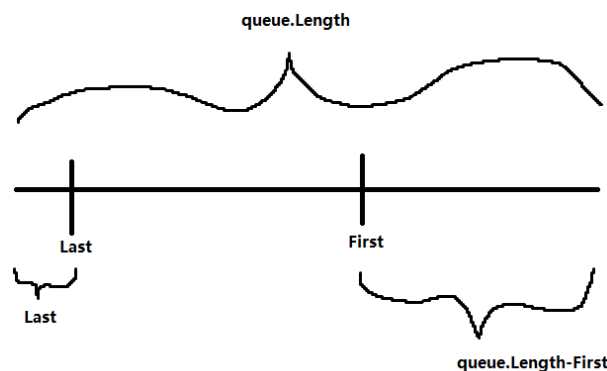
Delopgave c

I denne del af opgaven skal vi komme ind på om vores implementation dækker alle specifikationer eller om der er særtilfælde hvor koden ikke virker. Udgangspunktet i opgaven er at køen skal bevæge sig i positiv

omløbsretning som vist i illustrationen ved enqueue 7,3, -1 og 5. Vi har valgt at implementere det i urets retning, da man som bruger i teorien ikke vil lægge mærke til det. Det har også gjort koden nemmere at forstå skulle man benytte den. Hvis koden skulle bruges til noget der krævede særlige videnskabelige metoder, ville man nok vælge en implementeringsløsning hvor den positive omløbsretning er overholdt som i f.eks cosinus og sinus.

I vores implementation af `dequeue` har vi også valgt at ændre selve værdien på det forreste element til `None` i stedet for at lade dem gamle `Some x` værdi forblive. Vi har gjort dette da det giver en bedre overskuelighed over forløbet og funktionerne, på trods af, at det måske ikke er helt det samme som tidligere illustreret. Hvis `Some x` værdien skulle forblive var vi nødsaget til at ændre vores `length`, så den ikke tæller alle `None`, men finder længden således:

- Anvende et `for-loop` fra `first` til `last` og tælle +1, hvor hvert element imellem dem.
- Hvis `first ≥ last` kunne vi finde længden ved sige `last + queue.Length - first`. Illustreret herunder.



En ting som vi ikke har håndteret er, hvis man forsøger at køre vores funktioner uden at køre `create` først. I disse tilfælde får man `F# error` koder, men følger man programmets struktur, burde der ikke komme fejl.

Delopgave d

For at sammeligne vores implementering af en cirkulær kø med en normal kø, opsummeres egenskaberne ved en normal kø hurtigt. Betragt den følgende array med 5 elementer `[1;2;3;4;5]`. Bemærk at denne array kan repræsentere en hvilken som helst fyldt kø. Hvis vi nu antager at denne array har egenskaberne fra en normal kø, kan vi ligesom for en cirkulær kø bruge `dequeue` til at fjerne de forreste elementer. Gør vi dette to gange får vi følgende array `[None;None;3;4;5]`. Ulempen ved en normal kø er de to forreste pladser nu er utilgængelige. Vi kan derfor ikke bruge `enqueue`, da køen kan tænkes som fortsat værende fyldt. Betragtes vores implementation af en cirkulær kø, kan man derimod tilføje flere elementer på de to nu tomme pladser. Tilføjes henholdsvis 6 og 7 får man nemlig `[6;7;3;4;5]`. Derfor er der ingen spildt plads. Denne forskel gør at en cirkulær kø er mere effektiv, da der benyttes mindre memory end for en normal kø.

En ulempe i vores implementation er at man ikke kan få fat i data der er blevet slettet, når først man har lavet `dequeue` på det. Derfor ville man skulle have en separat datastruktur (eller array) til at holde øje med de dequeuet elementer eller eventuelt gemme dem et sted.