Om behandling av tekststrenger i Go

Problem:

Gitt en tekstfil med en sekvens av tegn på en linje. Sekvensen er en kodet melding av en norsk tekst. Hvert tegn (en byte) er representert med dets heksadesimale verdi fra en av ISO/IEC 8859 kodesettene, og med en prefiks `\x` (forstavelse). Dermed representerer tekststreng `\xNN` en spesiell format, som brukes i Golang, hvor `NN` er et heksadesimalt tall mellom `00` (desimalt 0) og `FF` (desimalt `15x16^1=240 + 15x16^0=15 => 255`). Det finnes andre prefikser i andre programmeringsmiljøer for å representere heksadesimale verdier i en tekststrening, som, for eksempel, `0x` i programmeringsspråket C.

Rob Pike beskriver tekststrenger som representerer heksadesimale tall i Golang som: " … a string literal (more about those soon) that uses the `\xNN` notation to define a string constant holding some peculiar byte values." (Se Golang bloggen https://blog.golang.org/strings "The Go Blog: Strings, bytes, runes and characters in Go", sist sett 2017-03-09).

Oppgaven går ut på å lese inn filen byte for byte i en datastruktur i Golang ("byte slice", for eksempel) og bearbeide data slikt at det kan skrives ut om en meningsfull tekst.

I dette konkrete tilfelle fremstår sekvensen av tegn (første 3 tegn/bytes) i filen som dette `\x48\x65\x6e ...`

Hvis vi antar at kodingen, som blir brukt er fra ISO/IEC 8859 kodesettet kombinert med Go's måte å presenterer heksadesimale verdier som en tekststreng, så representerer, for eksempel, `\x48` (desimalt `4x16+8x1=72`) bokstaven "H" i alle varianter av ISO/IEC kodesettet, siden den tilhører den opprinnelige ASCII delen av kodesettet med desimalverdier for koder opp til 127.

72	110	48	01001000	H	&#72;		Uppercase H

Hvis man manuelt kopierer tekststrengen fra filen (treasure.txt) inn i Golang koden (kildekode, treasuere.go), så kan man analysere denne med flere verktøy fra Golang's formatterings pakke `fmt`, slik som Rob Pike eksemplifiserer i den nevnte bloggen:

```
const sample = "\x48\x65\x6e"
for i := 0; i < len(sample); i++ {
    fmt.Printf("%x ", sample[i])
}
```

Utskriften fra denne koden blir `48 65 6e`, da ved å kopiere teksten inn i Golang kildekode direkte og lagre filen, blir den konvertert til Golang's "string literal" og behandlet med Golang's innebygde funksjonalitet, som tolker[1] `\x` som ett tegn med en spesiell funksjon, - indikere at de to neste tegn representerer heksadesimale tall.

Hvis vi derimot leser innholdet i filen inn i et kjørende Go-program, vha. av funksjoner fra `os` og `io/ioutil` pakkene (for eksempel med denne koden http://www.devdungeon.com/content/working-files-go#read_all), tolkes **`\x`** som to ordinære tegn, **`\`** og **`x`**, fra ASCII kodesettet (med heksadesimale verdier **`5c`** og **`78`**, dvs. "Backslash" og "Lowercase x").

[1] eng. "parse" brukes for å beskrive en handling hvor et dataprogram analysere data og oversetter fra en kodet "byteslice" til en annen

Hvis vi så forsøker å skrive ut innholdet i datastrukturen, som er blitt fullt opp fra innholdet i filen treasure.txt, med følgende kode:

```
...
// bruker koden fra http://www.devdungeon.com/content/working-files-go#read_all
// for å lese innholdet inn i variabelen data av typen []byte og
// lager en løkke over de første elementene (opp til en 12 bytes, siden
// vår sekvens \x48 er blitt til 4 bytes 5c 78 34 38, som da representerer tegn
// \ x 4 og 8 fra ASCII kodesettet)

                    for i := 0; i < len(data[0:12]); i++ {
                        fmt.Printf("%x ", data[i])
                    }
```

får vi følgende resultat

```
                    5c 78 34 38 5c 78 36 35 5c 78 36 65
```

istedenfor

```
                    48 65 6e
```

Dette er et godt eksempel på at en kode, som representerer samme tekst, tolkes forskjellig i forskjellige kontekster. I konteksten av “string literal”, har \x en spesialmening for de aktuelle Go funksjonene, og sekvensen er hardkodet (dvs. definert før utførelsen av programmet). I en kontekst av innlesing av data fra en fil, derimot, behandles all data som ordinære bytes fra ASCII kodesett.

Man kunne tenke seg at, hvis man leser data inn fra en fil i “byte slice” og så endrer type til “string”, så kan man vise (dekode) innholdet med funksjonene (Printf, Println osv.) fra Golang’s fmt pakke. Men da ignorerer man følgende:

- `fmt.Printf(“%s”, string(byteslice_fra_fil))` “ser” ikke det samme som `fmt.Printf(“%s”, “\x48\x65\x6e”)` “ser”
- den første ser en string `\\x48\\x65\\x6e`, da vår filbehandlingsfunksjon http://www.devdungeon.com/content/working-files-go#read_all ikke har gitt \x en spesial mening

Dette kode eksemplet bør illustrere det (`data` er en “byte slice” med bytes fra filen treasure.txt, mens `sample` er en hardkodet string i Golang’s kildekode):

```
 1 var limit int = 116
 2 fmt.Printf("Data as hex x: %x\n", data[0:limit])
 3 fmt.Printf("Data as string s: %s\n", data[0:limit])
 4 fmt.Printf("Data as string q: %q\n", data[0:limit])
 5 fmt.Printf("Data as string +q: %+q\n", data[0:limit])
 6
 7 const sample =
"\x48\x65\x6e\x72\x69\x6b\x20\x41\x72\x6e\x6f\x6c\x64\x20\x57\x6
5\x72\x67\x65\x6c\x61\x6e\x64\x20\x28\x66\xf8\x64\x74"
 8 fmt.Printf("Sample as hex x: %x\n", sample)
 9 fmt.Printf("Sample as string s: %s\n", sample)
10 fmt.Printf("Sample as string q: %q\n", sample)
11 fmt.Printf("Sample as string +q: %+q\n", sample)
```

Linje 1 definerer et heltall, slik at man kun kan behandle de første limit bytes fra filen (for å få bedre oversikt)
Resultater fra linje 2 er de heksadesimale verdier for hver av de 116 bytes i treasure.txt

```
Data as hex x:
5c7834385c7836355c7836655c7837325c7836395c7836625c7832305c7834315c7837325c7836655c78
36665c7836635c7836345c7832305c7835375c7836355c7837325c7836375c7836355c7836635c783631
5c7836655c7836345c7832305c7832385c7836365c7866385c7836345c783734
```

Funksjonskall på linje 3 returnerer utskrift av disse bytes, dvs. \ og x er en del av tekststrengen:

```
Data as string s:
\x48\x65\x6e\x72\x69\x6b\x20\x41\x72\x6e\x6f\x6c\x64\x20\x57\x65\x72\x67\x65\x6c\x61
\x6e\x64\x20\x28\x66\xf8\x64\x74
```

Linjene 4 og 5 resulterer i Golang's sin interpretasjon av hva som er i tekststrengen. Vha. formatteringsverben %q[2], kan man se at \ må dobbles for å unngå mistolking i forhold til den spesielle funksjonen, som dette tegnet har i Golang's kilkdekode:

```
Data as string q:
"\\x48\\x65\\x6e\\x72\\x69\\x6b\\x20\\x41\\x72\\x6e\\x6f\\x6c\\x64\\x20\\x57\\x65\\x
72\\x67\\x65\\x6c\\x61\\x6e\\x64\\x20\\x28\\x66\\xf8\\x64\\x74"
Data as string +q:
"\\x48\\x65\\x6e\\x72\\x69\\x6b\\x20\\x41\\x72\\x6e\\x6f\\x6c\\x64\\x20\\x57\\x65\\x
72\\x67\\x65\\x6c\\x61\\x6e\\x64\\x20\\x28\\x66\\xf8\\x64\\x74"
```

På linje 7 er deler av innholdet fra tekstfilen kopiert direkte inn i kildekoden, dvs. omformattert fra en rå sekvens av bytes til noe, som Golang tolker på spesiell måte (blant annet de sammensatte tegn `\x`, `\n` og `\t`, som da representerer spesielle formatterings funksjoner i Golang).

Linje 8 viser da noe helt annet en den samme sekvense lest direkte fra en fil. Alle \x har forsvunnet, siden de hadde en spesiell mening i kildekoden:

```
Sample as hex x: 48656e72696b2041726e6f6c642057657267656c616e64202866f86474
```

Linje 9 er interessant, da man her får se noe av den meningsfulle (for mennesker som kan norsk) teksten som ligger bak koden. ? fremkommer pga. at %s formatteringsverben i Golang er ikke implementert for å skille mellom koder fra de forskjellige ISO/ICE varianter av kodesett. Den kan kun rekognisere UTF-8 representasjon av tegn. Siden `ø` er representer med `f8` i filen (`48656e72696b2041726e6f6c642057657267656c616e64202866f86474`) klarer ikke denne funksjonen å finne korrekt tegn for denne koden:

```
Sample as string s: Henrik Arnold Wergeland (f?dt
```

Linjene 10 og 11 illustrerer hvordan %q formatteringsverben forsøker å hjelpe utvikler, og viser frem den heksadesimale verdien som ligger i filen og som ikke er "forsåelig". `\xf8` er selvsagt Go's syntaks for å representere heksadesimale verdier:

```
Sample as string q: "Henrik Arnold Wergeland (f\xf8dt"
Sample as string +q: "Henrik Arnold Wergeland (f\xf8dt"
```

Man kan fort vise, at hvis man erstatter f8 i den heksadesimale tekststrenger sample med UTF-8 verdi for ø, får man ønsket utfall fra Golang. Utskriften fra linjene 8 til 11 er da følgende:

```
Sample as hex x: 48656e72696b2041726e6f6c642057657267656c616e64202866c3b86474
```

[2] %+q viser alle Unicode tegn, som Unicode "code points", men det er ingen i vårt tekststreng i dette tilfelle

```
Sample as string s: Henrik Arnold Wergeland (født
Sample as string q: "Henrik Arnold Wergeland (født"
Sample as string +q: "Henrik Arnold Wergeland (f\u00f8dt"
```

Men spørsmålet gjennstår, - hvordan kan vi omdanne inn-data fra fil, slik at vi kan tolke koden korrekt, dvs. få frem noen meningsfullt for et menneske som forstår det norske språket?

Det er mange måter å gjøre det på, og jeg vil illustrere her en av disse. Dere er velkommen og herved opppmuntret å prøve å finne andre løsninger.

Siden \x har en spesialbetydning, for å kunne finne de "gjemte" byte-ene, må vi finne disse i vårt "byte slice" og dra ut kun de tegn som ikke er `\` eller `x`. En mulig scenario er å laste hele innholdet fra filen treasure.txt inn i "byte slice" (som implisitt betyr i RAM/virtuelt minne), og så bearbeide denne "byte slice"-en med egnede funksjoner.

Et par funksjoner i pakken `strings` kan hjelpe oss her (jeg søkte etter "golang replace string" med Google's søkemotor):

```
r := strings.NewReplacer("\\x", "")
mods := r.Replace(string(byteslice))
```

strings.NewReplacer er definert her https://golang.org/pkg/strings/#NewReplacer og definerer hva som skal erstattes med hva. Her definerer man at `\x` i tekststrengen fra filen treasure.txt, skal erstattes med en tom tekststreng. Siden `\` har en spesiell mening for programmet (kompilator), som tolker Go-kode, må man bruke `\\` for å kunne definere "Backslash" i Go's "string literal".

Vi bruker `r`, som er en variabel av typen Replacer, for å utføre bytte (https://golang.org/pkg/strings/#Replacer.Replace).

Variabelen `r` er definert som den såkalte pekermottaker, som er en måte å definere metoder for typer (klasser i Java) i Golang. Som man kan se, kaller man da en metode `Replace` på typen `Replacer`. Slike konstruksjoner er meget utbredt i Golang. Du kan lese om og prøve ut pekermottakere her https://tour.golang.org/methods/4.

I funksjonsspesifikasjonen ser man at Replace funksjonen returnerer resultatet i en datastruktur av typen `string`. `mods` vil inneholde dette resultatet. Funksjonen også har et inn-data argument av typen `string`. Derfor bytter vi typen fra `[]byte` til `string` av vår `byteslice` som holder innholdet av filen treasure.txt i det primære minnet.

Etter at Replace har gjort jobben, kan vi sjekke innholdet i resultat-strengen med

```
fmt.Printf("%+q", mods)
fmt.Println(len(mods))
```

%+q er en meget bra formatteringsverb, siden den alltid tolker strengen i forhold til UTF-8 kodingen, og hvis den ikke lykkes, prøver den å erstatte "ukjente" tegn med heksadesimale verdier.
I dette tilfelle får returneres følgende tekststreng:

```
"48656e72696b2041726e6f6c642057657267656c616e64202866f864742031372e206a756e692031383
0382c2064f8642031322e206a756c692031383435290a5669206572652065e206e61736a6f6e2076692
06d65642c0a20766920736de520656e20616c656e206c616e67652c0a206574206665647265c616e642
076692066727964657320766 42c0a206f672076692c20766920657265206d616e67652e0a2056e5727
420686a65727465207665742c2076e5727420f87965207365720a2068766f7220676f6474206f6720766
16b6b657274204e6f7267652065722c0a2076e5722074756e6765206b616e20656e2073616e6720626c6
```

```
16e7420666c65720a206176204e6f7267657320e67265732d73616e67652e0a0a204d6572206772f86e7
4206572206772657373657420696e67656e73746564732c0a206d65722066756c6c7420617620626c6f6
d737465722076657665740a20656e6e206920646574206c616e642068766f72206a6567207469 6c66726
564730a206d656420666172206f67206d6f7220686172206c657665742e0a204a65672076696c2064657
420656c736b652074696c206d696e2064f8642c0a20656920627974746520646574206876 6f72206a656
72065722066f864642c0a206f6d206d616e20657420706172616469732 06d65672062f8640a206176207
0616c6d6572206f766572737 6657665742e0a\n"
```

Det som er viktig med denne strengen, er slutten av den. Vi ser at den inneholder en linjeskift tegn `\n`, som er LF (LineFeed) og som var grunnen at kallet til `hex.DecodeString` funksjonen, fra Golang's pakke encoding/hex forårsaket en feilmelding "**encoding/hex: odd length hex string**".
Se mer om `DecodeString` her https://golang.org/pkg/encoding/hex/#DecodeString

Lengden til mods er på 1045 bytes.

Hvordan få vekk kode for linjeskift, som `hex.DecodeString(mods)` klarer ikke å dekode?

Jeg søkte på "golang trim line feed" med Google's søkemotor og det ledet meg til en funksjon i pakken strings, - `TrimSuffix` https://golang.org/pkg/strings/#TrimSuffix

Ved å utføre kall på variabelen mods og samtidig overskrive denne variabelen med resultatet etter `TrimSuffix`, er ikke \n lenger i strenger;

```
mods = strings.TrimSuffix(mods, "\n")
fmt.Printf("%+q", mods)
fmt.Println(len(mods))
```

Resultater er en lengde på 1044 bytes og følgende slutten på strengen (viser bare få siste bytes for bedre oversikt):

```
"....657665742e0a"
```

Nå kan vi gjøre DecodeString, og returnere det foreløpige resultatet i variabelen decoded:

```
decoded, err := hex.DecodeString(mods)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s\n", decoded)
```

Så overlater jeg til dere å finne ut hva utskriften fra siste linjen blir. Oppgaven er ikke ferdig med dette, men man kan sikkert prøve å erstatte alle ISO/ICO tegn fra utvidet ASCII med UTF-8 kode ved å teste funksjonen Replace.

Det finnes også en annen relevant funksjon i pakken `bytes`, - `bytes.Replace` https://golang.org/pkg/bytes/#Replace, som er meget godt egnet for oppgaven.

SLUTT.
/JG