Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

September 23, 2020

0.1 Peg Solitaire

0.1.1 Lærervejledningn

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

0.1.2 Introduktion

Peg Solitaire er et spil hvor en person skal forsøge at fjerne pinde i et bræt med huller ved at foretage en række træk indtil der kun er en pind tilbage på brættet. I hvert træk kan en pind fjernes ved at en nabopind flyttes over pinden til et ledigt hul. For hvert træk efterlades brættet med en pind færre.

I den klassiske engelske version af spillet Peg Solitaire består brættet af 33 huller, som til at starte med er fyldt med 32 pinde; det midterste hul i brættet er ikke udfyldt og opgaven består i at det netop er det midterste hul, der til slut skal indeholde en pind.

I denne opgave skal der arbejdes mod at få computeren til at finde en løsning til den engelske version af Peg Solitaire. En løsning vil bestå i at computeren udskriver de træk der skal flyttes. Opgaven er delt i tre dele. I den første delopgave arbejdes der mod at implementere et modul til at repræsentere en brætkonstellation samt operationer til at foretage flytninger og derved danne nye brætkonstellationer. I den anden delopgave skal der arbejdes mod at gøre det muligt for en spiller at spille spillet ved brug af `fsharpi` således at computeren tillader at der foretages træk hvorefter den nye brætkonstellation udskrives. I den tredie delopgave skal der skrives en algoritme, som returnerer en liste af træk, der efterlader en enkelt pind i midten af brættet.

0.1.3 Opgave(r)

0.1.1: Modulet `Board`.

Hvert hul i brættet er identificeret ved en position (r,c), hvor r er rækken for hullet (se billedet til højre) og c er kolonnen hullet optræder i. Således er positionen for den tomme plads i midten $(3,3)$.
I det følgende skal vi benytte os af 64-bit heltal til at indeholde en komplet brætkonstellation (vi gør kun brug af de 49 mindstbetydende bit).

	0	1	2	3	4	5	6
0			●	●	●		
1			●	●	●		
2	●	●	●	●	●	●	●
3	●	●	●		●	●	●
4	●	●	●	●	●	●	●
5			●	●	●		
6			●	●	●		

I F# kan et bræt således repræsenteres ved brug af typen `uint64`, der repræsenterer (unsigned) 64-bit heltal:

```
type b = uint64
```

I den første del af opgaven ønskes der implementeret en række funktioner til at operere på brætkonstellationer. Funktionerne ønskes implementeret i et modul `Board`, som vil kunne

bruges både af en rigtig spiller til at spille spillet og af et modul der har til hensigt at finde en løsning til spillet.

Modulet `Board` skal indeholde følgende typer og funktioner:

```
type b                                  // board type
type pos = int * int                    // position type
type dir = Up | Down | Left | Right     // move direction
type mv = pos * dir                     // move

val init     : unit -> b                // initial board
val valid    : pos -> bool              // is the position valid?
val peg      : b -> pos -> bool         // true if pos valid and
                                        //  hole contains a peg
val mv       : b -> mv -> b option      // returns new board
val pegcount : b -> int                 // number of pegs
val print    : b -> string              // string representation
```

Her følger nogle gode råd til hvordan ovenstående modul implementeres:

- Start med at implementere to hjælpefunktioner `seti` og `geti` til henholdsvis at sætte en givet bit i en `uint64`-værdi samt at undersøge om en givet bit er sat (hertil skal I benytte et udvalg af bit-operationer, inklusiv `|||`, `&&&`, `~~~`, `>>>` og `<<<`).
- Implementér en hjælpefunktion `posi` til at omdanne en position (row-column pair) til et bit-index i brætrepræsentationen.
- Funktionen `valid` skal returnere `false` hvis positionen ikke repræsenterer en hul-position i et tomt bræt.
- Implementér en funktion `neighbor` af type `pos -> dir -> pos option`, som, givet en valid position og en retning, returnerer en valid naboposition, hvis en sådan findes i den specificerede retning, eller værdien `None`. Et kald `neighbor(2,5)Right` skal returnere værdien `None` og et kald `neighbor(3,5)Right` skal returnere værdien `Some(3,6)`.
- Funktionen `mv` kan nu implementeres ved brug af funktionerne `peg`, `neighbor`, `seti` og `posi`.
- For at implementere funktionen `print` kan der benyttes to nestede rekursive funktioner (eller to nestede for-løkker), som hver itererer over henholdsvis rækkerne og kolonnerne på brættet.

I rapporten skal I beskrive jeres designovervejelser og demonstrere at jeres implementation fungerer som forventet (skriv unit-tests for de implementerede funktioner).

0.1.2: Modulet `Game`.

Implementér et modul `Game` med følgende funktionalitet:

```
val start : unit -> unit
val mv    : Board.mv -> unit
```

Modulet skal indeholde en reference til en "nuværende" brætkonstellation. Funktionen `start` skal sætte den nuværende brætkonstellation til det initielle bræt og udskrive brættet. Funktionen `mv` skal foretage en flytning (hvis det er lovligt) og udskrive den nye brætkonstellation.

I rapporten skal I demonstrere brugen af modulet `Game` i `fsharpi`.

0.1.3: Modulet `Solve`.

I denne delopgave skal der skrives en funktion `solve`, som foretager en udtømmende søgning efter en række træk som vil efterlade brættet i en konstellation med kun en pind, placeret i midten af brættet. Funktionen kan passende have følgende type:

```
type state = Board.b * Board.mv list
val solve : state -> Board.mv -> state option
```

Her består en tilstand af "den nuværende brætkonstellation" samt en liste af de træk der leder frem til denne konstellation (med det seneste træk forekommende først i listen). Funktionen tager yderligere et træk som argument. Ved at det gøres muligt at iterere gennem alle mulige træk (for hver konstellation), fra det første træk `((0,0),Right)` til det sidste træk `((6,6),Up)`, kan vi sikre at alle træk prøves. Funktionen skal benytte sig af rekursion til at foretage den (muligvis) udtømmende søgning. Givet en vilkårlig brætkonstellation samt en kandidat til et træk kan funktionen `solve` undersøge om trækket vil efterlade brættet i en ny konstellation eller om trækket ikke er gyldigt. Afhængigt af udfaldet kan det enten undersøges (ved eventuelt rekursivt at kalde `solve`) om den nye brætkonstellation har (eller er) en løsning eller om vi har bedre held med det næste træk i trækordningen (hvis et sådan træk findes).

For at implementere funktionen er det nyttigt først at implementere nogle hjælpefunktioner:

(a) Skriv en funktion `nextdir` af type `dir -> dir option`, som "roterer" en retningsværdi således at `Up` bliver til `Some Right`, `Right` bliver til `Some Down`, `Down` bliver til `Some Left` og `Left` bliver til `None`.

(b) Skriv en funktion `nextpos` af type `pos -> pos option`, som returnerer den næste position på et 7×7 hullers bræt (row-major). Et kald `nextpos(2,5)` skal returnere værdien `Some(2,6)` og et kald `nextpos(1,6)` skal returnere værdien `Some(2,0)`.

(c) Skriv en funktion `nextmv` af type `mv -> mv option`, som passende benytter sig af de to ovenfor specificerede funktioner. Funktionen skal give mulighed for at iterere gennem alle mulige flytninger, startende med flytningen `((0,0),Up)`. Bemærk at funktionen skal operere uden hensyn til en konkret brætkonstellation og at funktionen ikke skal tage højde for de præcise forekomster af huller i brættet (flytningerne kan senere filtreres blandt andet ved brug af funktionen `valid`). Således skal et kald `nextmv((1,2),Down)` returnere værdien `Some((1,2),Left)`, et kald `nextmv((1,6),Left)` skal returnere værdien `Some((2,0),Up)`. Endelig skal kaldet `nextmv((6,6),Left)` returnere værdien `None`.

I rapporten skal I vise koden for jeres implementation af den rekursive funktion `solve` og argumentere for at den finder en løsning til brætspillet, såfremt en sådan findes. Skriv også kode til at udskrive de fundne træk og vis i rapporten at jeres implementation finder en løsning til spillet i form af en liste af træk.

Rapporten skal også indeholde en beskrivelse af implementationens begrænsninger samt en reflektion over hvordan implementationen kan generaliseres til at finde løsninger til andre brætspecifikationer.