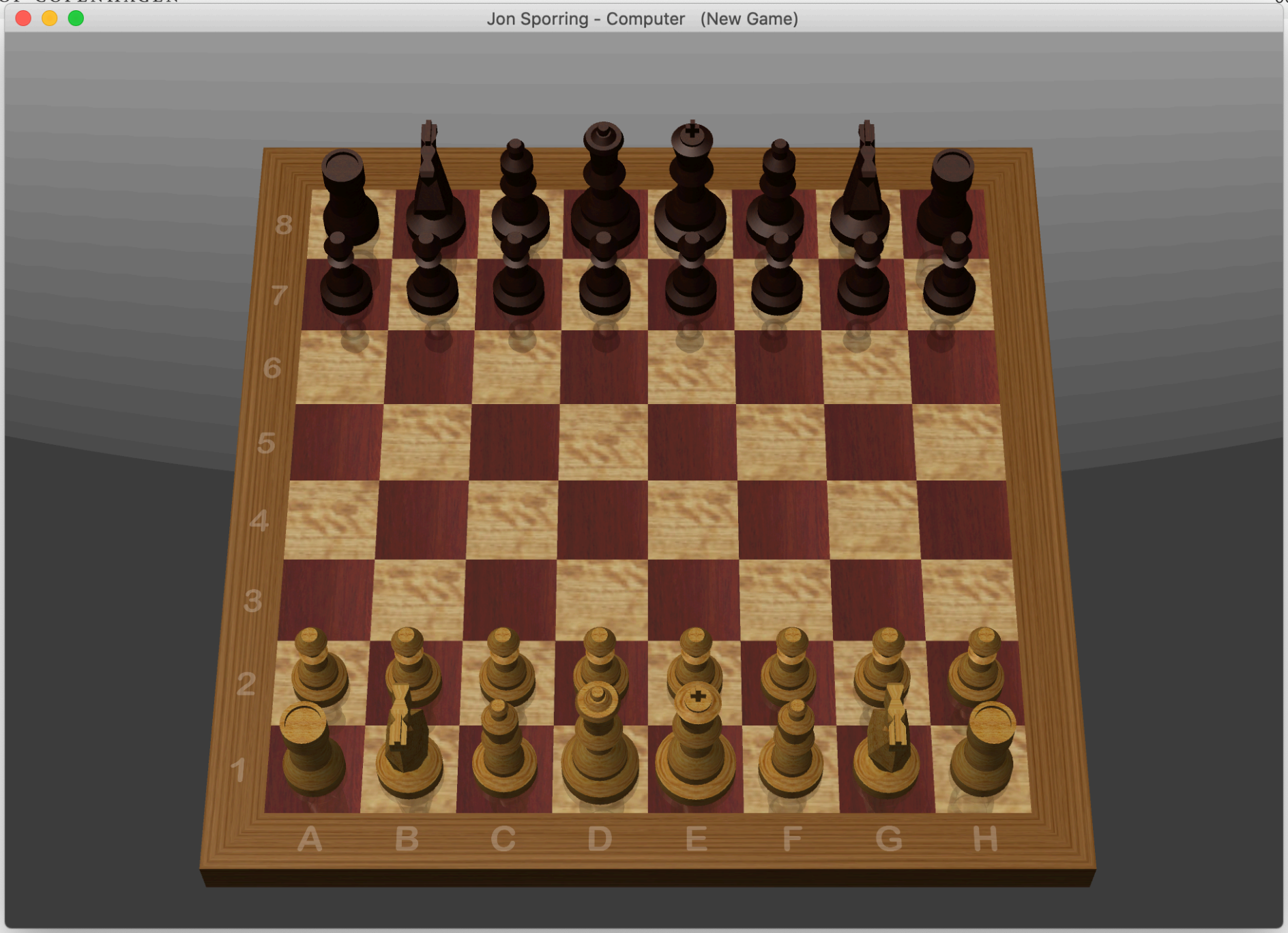
Introduktion til Programmering og Problemløsning (PoP)

Jon Sporring
Department of Computer Science
2021/01/03

UNIVERSITY OF COPENHAGEN

Skak



Hvordan bygger man en skakcomputer?

Elementer:

1. Spil-loop
2. En stilllingsbedømmelsesfunktion – vurderer stillingens fordel for hhv. sort og hvid
3. Et søgetræ

```
121 /// Play n turns starting with a given board and a given color
122 let rec play (b : board) (n : int) (c : Color) : board =
123   printfn "Board:\n%A\nScore: %A" b (b.score ())
124   match n with
125     0 ->
126         b
127     | _ ->
128         printfn "Turns left:%d\nPlayer: %A" n c
129         let newB = takeTurn b c
130         let nextC = opposingColor c
131         play newB (n-1) nextC
132
133 let initialBoard = createGame ()
134 let finalBoard = play initialBoard 4 White
```

En løkke f.eks. rekursion

Kommunikation med brugeren

En metode til at udføre et træk

Initialisering og opsætning

Administration

Elementer:

1. Stillingsbedømmelse: simpel (f.eks. sum af brikværdier) eller advanceret (f.eks. neuralt netværk trænet per ekspert)
2. Datastruktur til at formulere hypoteser i

type chessPiece

```
24 /// <summary> An abstract chess piece. </summary>
25 /// <param name = "col"> The color black or white </param>
26 [<AbstractClass>]
27 type chessPiece(color : Color) =
28   let mutable _position : Position option = None
29
30   /// The type of the chess piece as a string, e.g., "king" or "rook".
31   abstract member nameOfType : string
32
33   /// Make a deep copy including a copy of an existing pieces. Must be
34   /// implemented in the inheriting class and cast to chessPiece for
35   /// this to work as an abstract member, since the abstract class has
36   /// no knowledge of any future inheritors
37   abstract member copy : unit -> chessPiece
38
39   /// The value of the piece. The king should have the highest value.
40   abstract member value : float with get
41
```

Reference-typer kræver copy-konstruktor for at ungå aliasing, f.eks. til afprøvning af virtuelle trækfølger. (Board skal have tilsvarende)

Værdiansættelse af brik uafhængigt af position og nabobrikker

Copy-constructor for board

Imperativ eller funktionsparadigme:

- Array2D er muterbar datastruktur:
  - Godt hvis man har mange ændringer og vil ungå at kopiere hele strukturen hver gang.
  - Godt da det passer med brættets geometri
  - Skidt da man ofte kommer til at bruge kræfter på indeksadministration
  - Skidt da Array2D modul mangler fold, filter, etc.

type board

```
117   /// Make a deep copy af a board including all the pieces on it
118   member this.copy () =
119     let b = board ()
120     let copyPiece i j p =
121       b.[i,j] <- Option.bind (fun (p : chessPiece) -> Some (p.copy ())) p
122     Array2D.iteri copyPiece _board
123     b
124
```

Mellemløsning: iteri håndterer indekseringsrodet, løsning kan dog godt bliver svær at læse

Score: Valgt simpel sumationsløsning: hvid tæller postiv og sort negativ

type chessPiece

copy-konstruktor skal upcaste, så vi kan kopiere lister af brikker uden bekymring

```
14    override this.copy () = king (col) :> chessPiece
15    override this.value = -1.0**(col |> Color2Int |> float)*1e4;
```

Kræver lidt knofedt at konvertere discriminated union types til tal

module Chess

```
2  /// The possible colors of chess pieces
3  type Color = White | Black
4
5  let Color2Int (c : Color) : int =
6    if c = White then 0 else 1
7
8  let opposingColor (c : Color) : Color =
9    if c = White then Black else White
10
```

type board

```
175   member this.score () =
176     let mutable s = 0.0
177     let defValue (p : chessPiece option) =
178         match p with
179             None -> 0.0
180           | Some e -> e.value
181     Array2D.iter (fun e -> s <- s + defValue e) _board
182     s
```

chessPiece' enkeltbriksværdi. Konge høj, tårn mellem bonde lav.

Savner virkelig fold/backfold, men rækkefølge er ikke så oplagt

Hvad så med Artificial Intelligence (AI, kunstig intelligens)?

chessApp

```
27 /// Make an optimal move for a given color. If the board is unchanged,
28 /// then now valid move is possible.
29 let takeTurn (b : board) (c : Color) : board =
```

```
111   // find a good move if possible
112   let m =
113     b.pieces ()
114     |> List.filter (fun p -> p.color = c)
115     |> scoreMoves b
116     |> pickMove c
117   match m with
118     None -> b
119     | Some (src,dst) -> printfn "move: %A->%A" src dst; move b src dst
120
121 /// Play n turns starting with a given board and a given color
122 let rec play (b : board) (n : int) (c : Color) : board =
123   printfn "Board:\n%A\nScore: %A" b (b.score ())
124   match n with
125     0 ->
126         b
127     | _ ->
128         printfn "Turns left:%d\nPlayer: %A" n c
129         let newB = takeTurn b c
130         let nextC = opposingColor c
131         play newB (n-1) nextC
132
133 let initialBoard = createGame ()
134 let finalBoard = play initialBoard 4 White
```

En pipeline:

1. Find alle brikkerne på brættet
2. Behold dem med farve c
3. Beregn og bedøm alle træk for farv
4. Vælg et af dem

Fejlsituation? Strategi = Option typer
Debugging: Massere af printfn

Led i pipelinen

```
111  // find a good move if possible
112  let m =
113    b.pieces ()
114     |> List.filter (fun p -> p.color = c)
115     |> scoreMoves b
116     |> pickMove c
117  match m with
118    None -> b
119     | Some (src,dst) -> printfn "move: %A->%A" src dst; move b src dst
```

type board

```
11  /// Concatenate all v-elements in an array2d of None | Some v
12  let Array2DChoose (arr : 'a option [,]) : 'a list =
13    let con e lst =
14      match e with
15        None -> lst
16      | Some v -> v::lst
17    let mutable lst = []
18    Array2D.iter (fun e -> lst <- con e lst) arr
19    lst

125   /// Get a list of all chess pieces on the board
126   member this.pieces () = Array2DChoose _board
```

Strategi:

1. Tænk i generelle funktioner, de kan måske bruges senere.
2. Angiv typerne, det hjælper med at fange fejl og holde tankerne rene.

Led i pipelinen

type board

```
4  type Move = Position * Position
5  type ScoredMove = Move * float
6  type Runs = Position list * chessPiece list
```

```
68   // Evaluate all moves for a list of chessPieces and evaluate their
69   // resulting score. The result is a single, flat list of
70   // ((src,dest),score) elements including all moveable pieces.
71   let rec scoreMoves (b : board) (pieces : chessPiece list) : ScoredMove list =
72     printfn "pieces: %A" pieces
73     match pieces with
74       [] -> List<ScoredMove>.Empty
75       | p :: rst ->
76         let srcOption = p.position
77         let moves =
78           match srcOption with
79             None -> []
80             | Some src ->
81               // Add the opponent's neighbouring pieces's positions
82               let choises = p |> b.availableMoves |> availabeMoves2Pos p.color
83               let posPairs = List.map (fun dst -> (src,dst)) choises
84               let evalScore pair =
85                 let tmpBoard = move b (fst pair) (snd pair)
86                 tmpBoard.score ()
87               let scores = List.map evalScore posPairs
88               List.zip posPairs scores
89         moves @ (scoreMoves b rst)
```

```
111  // find a good move if possible
112  let m =
113    b.pieces ()
114    |> List.filter (fun p -> p.color = c)
115    |> scoreMoves b
116    |> pickMove c
117  match m with
118    None -> b
119    | Some (src,dst) -> printfn "move: %A->%A" src dst; move b src dst
120
```

Navngiv egne typer: det støtter den semantiske forståelse af strukturer (og gør linjer kortere)

Nogle gange kan det være en hjælp at bruge type som en del af navnet, men overdriv ikke!

Resultat af availableMoves homogeniseres: Nabobrikker i egen farve fjernes og modstanders farve erstattets deres position.
Valg: alle mulige træk samles i een liste af ScoredMove – bruger mere plads, men gør behandlingen mere homogen

Virtuelt bræt oprettes og slettes igen!

Led i pipelinen

type board

```
4  type Move = Position * Position
5  type ScoredMove = Move * float
6  type Runs = Position list * chessPiece list
```

```
111  // find a good move if possible
112  let m =
113    b.pieces ()
114     |> List.filter (fun p -> p.color = c)
115     |> scoreMoves b
116     |> pickMove c
117  match m with
118    None -> b
119     | Some (src,dst) -> printfn "move: %A->%A" src dst; move b src dst
```

```
91   // Pick an optimal 1-step ahead move
92   let pickMove (c : Color) (moves : ScoredMove list) : Move option =
93     printfn "moves: %A" moves
94     // White wants to maximize score, black minimize
95     let optVal =
96       if c = White then
97         tryMaxScore moves
98       else
99         tryMinScore moves
100    printfn "optVal: %A" optVal
101    // list of all equally good moves
102    let goodMoves = filterByScores moves optVal
103    printfn "goodMoves: %A" goodMoves
104    if goodMoves.Length > 0 then
105      let idx = rnd.Next goodMoves.Length;
106      let ((src, dst), score) = goodMoves.[idx]
107      Some (src,dst)
108    else
109      None
```

Konsekvens af valg: Hvid ønsker at maksimere. Skal vi abstrahere yderligere?

Der kan sagtens være flere lige gode optimale valg. Ekstra lag af udvælgelsesstrategi kræves

Søgetræsudfordring: Skal man have en model af modstanderens strategi?

Eksempelkørsel

15Chess — -zsh — 111×37

```
sporring@Jons-mac 15Chess % fsharpc chess.fs pieces.fs chessApp.fsx && mono chessApp.exe
Microsoft (R) F# Compiler version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.
Board:
7
6
5
4   k
3   r
2
1   R
0 K
  0 1 2 3 4 5 6 7
Score: 0.0
Turns left:4
Player: White
pieces: [R; K]
pieces: [K]
pieces: []
moves: [(((1, 1), (2, 1)), 0.0); (((1, 1), (0, 1)), 0.0); (((1, 1), (1, 2)), 0.0);
 (((1, 1), (1, 3)), 0.0); (((1, 1), (1, 4)), 0.0); (((1, 1), (1, 5)), 0.0);
 (((1, 1), (1, 6)), 0.0); (((1, 1), (1, 7)), 0.0); (((1, 1), (1, 0)), 0.0);
 (((1, 1), (3, 1)), 10.0); (((0, 0), (0, 1)), 0.0); (((0, 0), (1, 0)), 0.0)]
optVal: Some 10.0
goodMoves: [(((1, 1), (3, 1)), 10.0)]
move: (1, 1)->(3, 1)
Board:
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
Score: 10.0
```

Resumé

Til denne forelæsning har jeg talt om:

- Elementer i en skakcomputer
- Diskuteret konkrete programmeringsløsninger
- Diskuteret generelle dilemmaer og råd til programmering
  - Imperativ versus funktionsprogrammering
  - Brug af egendefinerede typer
  - Valg af værdinavne
  - Fejlhåndteringsstrategier
  - Valg af abstraktionsniveau

Spørgetime