BSWU BPRD **IT University, E2013**

Skriftlig eksamen, Programmer som Data
Prøveeksamen, oktober 2013

Version 1.0 af 2013-10-07

Dette eksamenssæt har 5 sider. Tjek med det samme at du har alle siderne.

Der er tale om en tag-med-hjem eksamen med en varighed på 29 timer.

Der er 4 opgaver. For at få fulde point skal du besvare alle opgaverne og deres delopgaver tilfredsstillende.

Du må bruge alle bøger, forelæsningsnoter, forelæsningsplancher, opgavesæt, dine egne opgavebesvarelser, internetressourcer, lommeregnere, computere, mobiltelefoner, og så videre.

Du må **naturligvis ikke plagiere** fra andre i din besvarelse. Din besvarelse må ikke indeholde tekst, programkode, figurer, tabeller eller lignende som er skabt af andre end dig selv, med mindre der er fyldestgørende kildeangivelse, dvs. at du beskriver oprindelsen af den pågældende tekst (eller lignende) på en komplet og retvisende måde. Det gælder også hvis den inkluderede kopi ikke er identisk, men tilpasset fra tekst eller programkode fra lærebøger eller fra andre kilder.

Hvis en delopgave kræver at du definerer en bestemt funktion, så må du gerne **definere alle de hjælpefunktioner du vil**, men du skal definere den ønskede funktion så den har netop den type og det resultat som delopgaven kræver.

Hvis der er uklarheder, inkonsistenser eller (formodede) fejl i opgaveteksten, så skal du i din besvarelse beskrive disse og beskrive hvilken tolkning af eksamenssættet du har anvendt ved besvarelsen af opgaverne.

Aflevering af besvarelsen

Besvarelsen skal bestå af programkode der kan køres samt en forklarende tekst (på dansk eller engelsk).

Hele besvarelsen skal afleveres som en zip-fil med navnet `BPRD-DitNavn.zip`, for eksempel `BPRD-JonasJensen.zip`. NB: Ingen gzip- eller bzip- eller rar- eller tar-filer, tak. Zip-filen skal indeholde en mappe kaldet `BPRD-DitNavn`, og filer i mappen skal have en af følgende formater:

- Ren tekst (*.txt)
- Programkode (*.fs, *.fsy, *.fsl)
- PDF (*.pdf)
- Billedfiler, kun PNG (*.png) eller GIF (*.gif) eller JPEG (*.jpg)

Zip-filen må ikke indeholde DLL eller EXE filer. Sådanne Zip-filer bliver nemlig stiltiende sorteret fra af mailserveren og kommer ikke igennem til modtageren, da de opfattes som en sikkerhedsrisiko.

Besvarelsen kan afleveres ved at sende den vedhæftet en mail til `sestoft@itu.dk`. [I praksis har jeg ikke tid til at bedømme besvarelserne på prøveeksamenssættet, men det er godt at teste om denne afleveringsform fungerer. Til den rigtige eksamen håber jeg vi får lov at bruge elektronisk aflevering. Det bliver noget med en eksakt tidsfrist, fredag 3. januar 2014 kl 14:00:00; nærmere oplysninger følger.] [Dette eksamenssæt dækker ikke kursets pensum ligeligt. Som forklaret i forelæsningen skal eksamenssættet afspejle stof som I har øvet i ugeopgaver, og det er jo indtil videre kun en mindre del af kurset.]

Baggrund for opgavesættet

Opgaven tager udgangspunkt i regulære udtryk og endelige automater som beskrevet i PLC kapitel 3 og Mogensen ICD afsnit 1.1-1.8, 2.1-2.5, 2.11, 2.12, 2.16 (eller BCD 2.1-2.7, 2.9, 3.1-3.6, 3.12, 3.17), samt forelæsning og tutorials.

Som forklaret i disse materialer kan elementære regulære udtryk `RE` beskrives ved følgende grammatik:

```
RE ::= 'char'
     | eps
     | RE RE
     | RE *
     | RE | RE
     | ( RE )
```

Opgave 1 (25 %): Lexerspecifikation og parserspecifikation

Opgave 1.1

Lav en lexerspecifikation `Re.fsl` til `fslex` for en lexer der kan læse regulære udtryk som beskrevet ovenfor.

Opgave 1.2

Lav en parserspecifikation `Re.fsy` til `fsyacc` for en parser der kan læse regulære udtryk som beskrevet ovenfor.

Husk at konventionen for regulære udtryk er at `*` binder stærkere end sekvens `RE RE`, og sekvens binder stærkere end alternativ `RE | RE`, så `'a' 'b' * | 'c'` er det samme som `('a' ('b' *)) | 'c'`.

Opgave 1.3

Regulære udtryk kan repræsenteres med denne abstrakte syntakstype `re` i F#:

```
type re = Char of char
        | Eps
        | Seq of re * re
        | Star of re
        | Choice of re * re
```

Definér en F#-funktion `parse : string -> re` der bygger på den lexer og parser der er konstrueret i de foregående opgaver.

Opgave 1.4

Dokumentér at funktionen `parse` virker korrekt på følgende eksempler på regulære udtryk (ét per linje):

```
'a' | 'b' *
'a'|'b'*
'a' * 'a' 'a'
'a' 'b' * | 'c'
( 'a' ('b' *) ) | 'c'
( ('a' 'b') * | 'b') *
```

Med andre ord, vis resultatet af at anvende funktion `parse` på hvert af disse eksempler og argumentér for at resultatet er korrekt.

Opgave 1.5

Skriv 20–40 linjers forklaring (hvor en linje som normalt er på 60–100 tegn) af hvordan dine lexer- og parser-specifikationer er opbygget, og af eventuelle særlige problemer der måtte løses for at få lexer, parser og funktionen `parse` til at virke.

Opgave 2 (20 %): Repræsentation af NFAer

Som bekendt består en ikke-deterministisk endelig automat (non-deterministic finite automaton, NFA) af en starttilstand, nul eller flere accepttilstande, og en transitionsrelation som er en mængde af tripler `(s, sym, t)` hvor `s` er fra-tilstand, `t` er til-tilstand, og `sym` er et symbol: enten den tomme sekvens ϵ (epsilon) eller et tegn.

For at forenkle arbejdet nedenfor antager vi at enhver NFA kun har én accepttilstand. (Det er altid muligt at opnå).

En sådan NFA kan repræsenteres i F# med denne type:

```
type state = int
type sym = SEps | SChar of char
type nfa =
  { start  : state;
    accept : state;
    trans  : (state * sym * state) list
  }
```

En tilstand `state` er blot et tal. Transitionsrelationen `trans` er en liste af alle maskinens tilstandsovergange, sådan at `(17, SChar 'a', 42)` betyder at man kan gå fra tilstand 17 til tilstand 42 på symbolet `'a'`, og `(42, SEps, 17)` betyder at man kan gå fra tilstand 42 til tilstand 17 på den tomme streng ϵ (epsilon).

Opgave 2.1

Skriv en F#-funktion `order : nfa -> nfa` der konverterer en given NFA til en mere overskuelig repræsentation hvor listen `trans` er sorteret så alle transitioner `(s, sym, t)` er ordnet leksikografisk efter fra-tilstanden `s` og symbolet `sym`. Det vil sige at alle transitioner fra samme tilstand kommer lige efter hinanden, og endvidere at alle transitioner fra samme tilstand og på samme symbol kommer lige efter hinanden.

Symbolerne skal ordnes sådan at `SEps` kommer før alle `SChar`, og sådan at `SChar c1` kommer før `SChar c2` hvis tegn `c1` kommer før tegn `c2`, og så videre.

Opgave 2.2

Skriv en F#-funktion `isDfa : nfa -> bool` der afgør om en given NFA også er en DFA, dvs. at den ikke har epsilon-transitioner, og at den ikke har to eller flere transitioner på samme symbol fra samme tilstand.

Opgave 2.3

Skriv 20-30 linjers forklaring af funktionerne `order` og `isDfa` og argumentér for deres korrekthed.

Opgave 3 (35 %): Generering af NFAer

Som bekendt kan man ud fra et regulært udtryk systematisk generere en ikke-deterministisk endelig automat (non-deterministic finite automaton, NFA) der genkender præcis de samme strenge som det regulære udtryk.

For at forenkle konstruktionen antager vi som ovenfor at enhver NFA kun har én accepttilstand, og benytter `nfa`-typen fra opgave 2.

Opgave 3.1

Skriv en F#-funktion `makeNfa : re -> nfa` der ud fra et givet regulært udtryk konstruerer en NFA der genkender de samme strenge som det regulære udtryk.

Vink: Standardalgoritmen fungerer ved at kombinere simplere NFAer til mere komplekse NFAer. For at undgå at skulle omnummerere tilstande kan det betale sig at generere NFAerne så den samme tilstand (der jo blot er et nummer) ikke indgår i mere end én af de simple NFAer. Dette kan opnås ved at benytte en global generator af nye tilstandsnumre, som fx funktionen `newState : unit -> int` nedenfor:

```
let nextState = ref -1
let newState () = (nextState := 1 + !nextState; !nextState)
```

Opgave 3.2

Skriv 20-40 linjers forklaring af din `makeNfa`-funktion, herunder argumenter for dens korrekthed.

Opgave 3.3

Vis resultaterne af `makeNfa(parse r)` for hver af de eksempler på regulære udtryk der er vist i opgave 1.4. Forklar de resulterende NFAer og argumentér for at de er korrekte.

Opgave 4 (20 %): Udvidelse af konkret syntaks

Ud over de elementære former for regulære udtryk vist på side 2 er der som bekendt en lang række afledte former, der alle kan udtrykkes ved hjælp af de elementære former. Her er nogle eksempler:

```
['a' 'e' 'i' 'o' 'u']              set of characters
['0'-'9']                          range of characters
['0'-'9' 'a'-'z']                  ranges of characters
('a' | 'b') ?                      zero or one occurrence
('a' | 'b') +                      one or more occurrences
['+' '-']? ['0'-'9']+              combinations of the above
```

Opgave 4.1

Skriv en ny lexerspecifikation `MoreRe.fsl` til fslex der understøtter de viste udvidede regulære udtryk.

Opgave 4.2

Skriv en ny parserspecifikation `MoreRe.fsy` til fsyacc der understøtter de viste udvidede regulære udtryk og genererer abstrakt syntaks af type `re` som i opgave 1.3. Bemærk at typen `re` ikke skal ændres.

Opgave 4.3

Kombinér den nye lexer og parser med en funktion `parse : string -> re` der genererer abstrakt syntaks af type `re` fra opgave 1.3.

Opgave 4.4

Dokumentér at funktionen `parse` virker korrekt på de eksempler på regulære udtryk der er vist først i opgave 4.

Med andre ord, vis resultatet af at anvende funktion `parse` på hvert af disse eksempler og argumentér for at resultatet er korrekt.

Vis også at der fortsat produceres korrekt abstrakt syntaks for eksemplerne fra opgave 1.4

Opgave 4.5

Skriv 20-40 linjers forklaring af hvordan dine lexer- og parserspecifikationer er opbygget, og af eventuelle særlige problemer der måtte løses for at få lexer, parser og funktionen `parse` til at virke.