BSPRDAT1KU **IT University, E2023**

Skriftlig eksamen, Programmer som Data
Januar 2024

Version 1.00 af January 6, 2024

Dette eksamenssæt har 11 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside mandag den 8. januar 2024 kl 14:00.
Besvarelsen skal afleveres elektronisk i LearnIt senest **tirsdag den 9. januar 2024 kl 14:00** som følger:

- Besvarelsen skal uploades på kursets hjemmeside i LearnIt under **Ordinary exam assignment**.
- Der kan uploades en fil, som skal have en af følgende typer: `.txt` eller `.pdf`. Hvis du for eksempel laver besvarelsen i LaTeX, så generer en pdf-fil. Hvis du laver en tegning i hånden, så scan den og inkluder det skannede billede i det dokument du afleverer.

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

Hvis der er uklarheder, inkonsistenser eller tilsyneladende fejl i denne opgavetekst, så skal du i din besvarelse beskrive disse og beskrive hvilken tolkning af opgaveteksten du har anvendt ved besvarelsen. Hvis du mener det er nødvendigt at kontakte opgavestiller, så send en email til `sap@itu.dk` med forklaring og angivelse af problem i opgaveteksten.

Din besvarelse skal laves af dig og kun dig, og det gælder både programkode, lexer- og parserspecifikationer, eksempler, osv., og den forklarende tekst der besvarer opgavespørgsmålene. Det er altså ikke tilladt at lave gruppearbejde om eksamen.

Din besvarelse skal indeholde følgende erklæring:

Jeg erklærer hermed at jeg selv har lavet hele denne eksamensbesvarelse uden hjælp fra andre.

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

Du må **naturligvis ikke plagiere** fra andre kilder i din besvarelse, altså forsøge at tage kredit for arbejde, som ikke er dit eget. 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 opgave kræver, at du definerer en bestemt funktion, så må du gerne **definere alle de hjælpefunktioner du vil**, men du skal definere funktionen så den har den ønskede type og giver det ønskede resultat.

Udformning af besvarelsen

Besvarelsen skal bestå af forklarende tekst (på dansk eller engelsk) der besvarer spørgsmålene, med væsentlige programfragmenter indsat i den forklarende tekst, eller vedlagt i bilag (der klart angiver hvilke kodestumper der hører til hvilke opgaver).

Vær omhyggelig med at programfragmenterne beholder det korrekte layout når de indsættes i den løbende tekst, for F#-kode er som bekendt layoutsensitiv.

Det forventes at løsninger suppleres med tests der demonstrerer at løsningerne fungerer efter hensigten.

Snydtjek

Til denne eksamen anvendes *Snydtjek*. Cirka 10% vil blive udtrukket af studieadministrationen i løbet af eksamen. Navne bliver offentliggjort på kursets hjemmeside tirsdag den 9. januar klokken 14:00. Disse personer skal møde op i det Zoom møde, som også opslås på kursets hjemmeside, sammen med de udtrukne navne. Man vil blive trukket ind til mødet en af gangen.

Til Snydtjek er processen, at hver enkelt kommer ind i 5 minutter, hvor der stilles nogle korte spørgsmål omkring den netop afleverede besvarelse. Formålet er udelukkende at sikre at den afleverede løsning er udfærdiget af den person, som har uploaded løsningen. Du skal huske dit studiekort.

Det er obligatorisk at møde op til snydtjek i tilfælde af at du er udtrukket. Udeblivelse medfører at eksamensbesvarelsen ikke er gyldig og kurset ikke bestået. Er man ikke udtrukket skal man ikke møde op.

Opgave 1 (20%) Icon

Kapitel 11 i *Programming Language Concepts* (PLC) introducerer "continuations" og "backtracking" samt sproget Icon. Koden der anvendes i dette spørgsmål findes i filen `Lectures/Lec11/Cont/Icon.fs` i kursets git repository.

I Icon kan vi f.eks. skrive `write(5 to 27)`. Ved at anvende implementationen i filen `Icon.fs` kan vi udtrykke dette i abstrakt syntaks:

```
let examEx1 = Write(FromTo(5,27))
```

og køre eksemplet, der udskriver tallet 5 på skærmen, inde fra F# fortolkeren:

```
> run examEx1;;
5 val it : value = Int 5
```

1. Skriv et Icon udtryk, som udskriver værdierne 5 6 7 8 9 10 11 12 på skærmen:

   ```
   > run ...;;
   5 6 7 8 9 10 11 12 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Forklar hvorledes du får udskrevet alle 8 tal.

2. Skriv et Icon udtryk, som udskriver alle tal n mellem 3 og 60, hvor 3 går op i tallet n:

   ```
   > run ...;;
   3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Det er et krav at `FromTo` indgår i din løsning, f.eks. `FromTo(1,20)`. Du skal forklare hvordan din løsning fungerer.

3. Skriv et Icon udtryk, som udskriver alle tal n mellem 4 og 61, hvor 3 går op i $n - 1$:

   ```
   > run ...;;
   4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar.

   Hint: Du kan med fordel tage udgangspunkt i løsningen til spørgsmål 2 ovenfor.

4. Udvid implementationen af Icon med en ny generator `RandomFromList(N,xs)`, som genererer *N* tilfældigt udvalgte heltal fra listen af heltal *xs*. Det antages, at $N > 0$ og *xs* er en ikke tom liste af heltal. Generatoren `RandomFromList(N,xs)` fejler med det samme, hvis $N \leq 0$ eller *xs* er den tomme liste.

   Eksempelvis giver

   ```
   > run (Write(RandomFromList(1,[42])));;
   42 val it : value = Int 42
   ```

   Nedenstående eksempel kan give tre andre udvalgte tal når du kører det.

   ```
   > run (Every(Write(RandomFromList(3,[1;2;3;4;5]))));;
   4 4 3 val it : value = Int 0
   ```

   Du kan bruge .NET klassen `System.Random` til at vælge tal tilfældigt fra *xs*. Vis koden for din implementation af `RandomFromList`.

5. Definer test eksempler, som tester relevante grænsetilfælde af din implementation af `RandomFromList(N,xs)`. Kør og vis uddata for alle dine test eksempler. Eksempelvis tester nedenstående, at der for *N* lig 0 ikke genereres et tal fra listen *xs*.

   ```
   > run (Every(Write(RandomFromList(0,[1;2;3;4;5]))));;
   val it : value = Int 0
   ```

Opgave 2 (25%) Micro-SML: Print statistik

Kapitel 13 i *Programming Language Concepts* (PLC) introducerer sproget micro–SML. Derudover introduceres en bytekode maskine i C, **msmlmachine.c**. Koden der anvendes her findes i kursets git repository under `Lectures/Lec13/MicroSMLpublic` og `Lectures/Lec13/MicroSMLpublic/MsmlVM/src`.

I denne opgave tilføjer vi et nyt udtryk `printStat` e, som udskriver to informationer omkring stakken på det tidspunkt `printStat` e udføres; stakkens størrelse og antallet af aktiveringsposter (eng. *stack frames*). Udtrykket e skal evaluere til et tal og anvendes til at kunne adskille forskellige kald til `printStat`.

Betragt nedenstående program, som du kan gemme i filen **printstat.sml**:

```
fun genList n =
  let
    fun loop n = fn acc ->
      (printStat 3;
       if n < 0 then acc else loop (n-1) (n::acc))
  in
    printStat 2;
    loop n nil
  end

begin
  printStat 1;
  genList 2
end
```

Programmet kan oversættes med og uden optimeringer:

```
% mono ./microsmlc.exe -opt printstat.sml
Micro-SML compiler v 1.1 of 2018-11-18
Compiling printstat.sml to printstat.out

Compiled to file printstat.out

% mono ./microsmlc.exe printstat.sml
Micro-SML compiler v 1.1 of 2018-11-18
Compiling printstat.sml to printstat.out

Compiled to file printstat.out
```

Flaget `-opt` angiver, at der oversættes med optimeringer. Nedenstående uddata i venstre kolonne er efter kørsel af det oversatte program med optimeringer. Højre kolonne er efter kørsel uden optimeringer.

```
-------PRINTSTAT 1-------      -------PRINTSTAT 1-------
  Size stack: 5                 Size stack: 5
  Number stack frames: 1        Number stack frames: 1
-------PRINTSTAT 2-------      -------PRINTSTAT 2-------
  Size stack: 9                 Size stack: 11
  Number stack frames: 1        Number stack frames: 2
-------PRINTSTAT 3-------      -------PRINTSTAT 3-------
  Size stack: 7                 Size stack: 15
  Number stack frames: 1        Number stack frames: 3
-------PRINTSTAT 3-------      -------PRINTSTAT 3-------
  Size stack: 7                 Size stack: 19
  Number stack frames: 1        Number stack frames: 4
-------PRINTSTAT 3-------      -------PRINTSTAT 3-------
  Size stack: 7                 Size stack: 23
  Number stack frames: 1        Number stack frames: 5
-------PRINTSTAT 3-------      -------PRINTSTAT 3-------
  Size stack: 7                 Size stack: 27
  Number stack frames: 1        Number stack frames: 6
```

Hvert kald til `printStat` vil printe `-----PRINTSTAT <e>-----`, hvor **<*e*>** er den værdi (tal), som udtrykket e evalueres til. Derefter følger stakkens størrelse og antallet af *stack frames*. Layout af stakken er beskrevet i afsnit 13.5.1 og figur 13.10 i PLC.

1. Det fremgår ovenfor, at der maksimalt er et *stack frame* når det optimerede program kører. Når det ikke optimerede program kører er der op til 6 *stack frames*. Forklar hvad der er årsagen til dette. Du skal være præcis i din argumentation, med reference til alle relevante steder i koden **printstat.sml**, for at få fuld point.

2. Du skal udvide lexer **FunLex.fsl** med support for `printStat` e. Du skal repræsentere `printStat` som et primitiv med 1 argument, dvs. `Prim1`. Således skal hverken **FunPar.fsy** eller **Absyn.fs** ændres.

   Byg oversætteren `microsmlc.exe` med din nye lexer og vis, at du får nedenstående uddata, når du oversætter med flaget `-verbose`:

   ```
   % mono ./microsmlc.exe -verbose printstat.sml
   Micro-SML compiler v 1.1 of 2018-11-18
   Compiling printstat.sml to printstat.out

   Program after alpha conversion (exercise):
   fun genList n =
     let
       fun loop n =
         fn acc -> (printStat(3) ;
                    if (n < 0) then acc else loop (n - 1) (n :: acc))
     in
       (printStat(2) ; loop n nil)
     end
   begin
     (printStat(1) ; genList 2)
   endParseTypeAndRun.compProg' ERROR: typ of Prim1 printStat not implemented
   %
   ```

   Funktionen `printStat` kan nu lexes og parses, men oversætteren fejler fordi type inferens for `printStat` ikke er implementeret.

3. Figur 13.6 på side 260 i PLC viser typeregler for micro–SML. Der henvises endvidere til figur 6.1 på side102. Nedenfor ses typereglen for for `printStat`.

   $$(\textit{printStat})\frac{\rho \vdash e : \texttt{int}}{\rho \vdash \texttt{printStat}\ e : \texttt{int}}$$

   Reglen kræver at e har typen `int` og i så fald vil hele udtrykket have typen `int`. Angiv et typetræ for udtrykket

   ```
   let val x = 42 in x + printStat 1 end
   ```

   Du finder to eksempler på typetræer i figur 4.8 og 4.9 på side 72 i PLC.

4. Implementer typereglen *printStat* for micro–SML i filen **TypeInference.fs**. Du skal blot udvide funktionen `typExpr` for tilfældet `Prim1(`*ope*`,`*e1*`,_)`, hvor *ope* er `printStat`.

   Byg oversætteren `microsmlc.exe` med din opdaterede **TypeInference.fs** fil og vis, at du får nedenstående uddata, når du oversætter med flaget `-verbose`:

   ```
   % mono ./microsmlc.exe -verbose printstat.sml
   ...
   Program with types:
   fun genList n =
     let
       fun loop n =
         fn acc -> (printStat(3:int):int ;
                    if (n:int < 0:int):bool
   ```

```
                    then acc:(int list)
                    else loop:(int -> ((int list) -> (int list)))
                           (n:int - 1:int):int:((int list) -> (int list))_tail
                             (n:int :: acc:(int list)):(int list):(int list))
                               :(int list):((int list) -> (int list))
  in
    (printStat(2:int):int ;
     loop:(int -> ((int list) -> (int list)))
       n:int:((int list) -> (int list))_tail
        nil:(int list):(int list)):(int list)
  end
begin
  (printStat(1:int):int ;
   genList:(int -> (int list))_tail 2:int:(int list)):(int list)
end
Result type: (int list)
ParseTypeAndRun.compProg' ERROR: cExpr.Prim1 printStat not implemented
%
```

Du skal forklare dine ændringer i **TypeInference.fs** for at få fuld point.

5. For at udskrive statistikken på køretid, implementerer vi en ny bytekode instruktion `PRINTSTAT` i **Machine.fs**:

```
type instr =
  | Label of label (* symbolic label; pseudo-instruc. *)
  ...
  | PRINTSTAT      (* PrintStat, Exam E2023           *)
```

Vis alle rettelser til **Machine.fs** således at bytekodeinstruktionen `PRINTSTAT` kan anvendes af oversætteren i **Contcomp.fs**.

6. Bytekode maskinen **msmlmachine.c** skal tilsvarende udvides med bytekode instruktionen `PRINTSTAT`:

	Instruction	**Stack before**		**Stack after**	**Effect**
0	`CSTI` i	s	$\Rightarrow$	s, i	Push constant i
...					
43	`PRINTSTAT`	s, v	$\Rightarrow$	s, v	Print stat using v in header: `-----PRINTSTAT <e>-----`

Den præcise formattering på skærmen er vist ovenfor med **printstat.sml** som eksempel.

Du skal implementere `PRINTSTAT` i **msmlmachine.c**. Husk, at beskrive din implementation.

Hint: En mulig tilgang er at starte med basepeger `bp` og løbe kæden af basepegere igennem indtil den første basepeger med værdien -999 findes. Da hver *stack frame* gemmer netop en gammel basepeger (eng. *old base pointer*) kan du tælle antallet af *stack frames*.

Hint: Du kan anvende nedenstående kodestruktur i **msmlmachine.c**:

```
int execcode(word p[], word s[], word iargs[], int iargc, int /* boolean */ trace) {
  word bp = -999;        // Base pointer, for local variable access
  ...
    switch (p[pc++]) {
    case CSTI:
      ...
    case PRINTSTAT: // Exam, E2023
      printStat(s[sp], s, bp, sp); break;
    default:
    ...
```

```
}

void printStat(word N, word s[], word bp, word sp) {
  printf("-------PRINTSTAT " WORD_FMT "-------\n", Untag(N));

  printf("  Size stack: " WORD_FMT "\n", ...);
  word bp_i = bp;
  word numFrames = 0;
  while (bp_i != -999) {
    ...
  }
  printf("  Number stack frames: " WORD_FMT "\n", numFrames);
  return;
}
```

Bemærk, at værdien af basepeger i en *stack frame* er tagget, dvs. du skal anvende makro `Untag` for at untagge værdien.

7. Du skal nu udvide implementationen af `Prim1` i filen **Contcomp.fs** til at generere kode for `printStat`. Oversætterskemaet ser således ud:

   $$\begin{aligned}&\mathrm{E}[\![\texttt{printStat}\ e]\!] = \\ &\qquad \mathrm{E}[\![e]\!] \\ &\qquad \texttt{PRINTSTAT}\end{aligned}$$

   Vis, at din implementation fungerer ved at inkludere den genererede bytekode og vise uddata ved kørsel af programmet **printstat.sml**. Du skal oversætte og køre programmet med og uden optimeringer, `-opt`, og dermed vise at du får samme resultat som ovenfor.

Opgave 3 (25%) Micro–ML: Køer

Kapitel 5 i *Programming Language Concepts* (PLC) introducerer evaluering af et højereordens funktionssprog. Koden der anvendes her findes i kataloget `Lectures/Lec05/Fun` i kursets git repository. Opgaven er at udvide funktionssproget med køer (eng. *queues*), således at vi kan evaluere udtryk der manipulerer køer, se eksempel `ex01` nedenfor:

```
let q1 = [1 -> 2 -> 3] in
  let q2 = [1 -> 4] in
    let q3 = 5 ->> q1 ++ q2 in
      <<- q3
    end
  end
end
```

Vi har tilføjet fire syntaktiske konstruktioner:

- En ikke tom kø: $[e_n\texttt{->}\ldots\texttt{->}e_1]$, $n \geq 1$. Det sidste element indsat er e_n og det næste element der tages ud af køen er e_1.
- En operator e_1 `++` e_2 som sætter to køer sammen repræsenteret ved udtrykkene e_1 og e_2. I eksemplet ovenfor bliver `q1 ++ q2` til køen `[1 -> 2 -> 3 -> 1 -> 4]`. Det næste tal der tages ud af køen er `4`. Operatoren `++` er venstre associativ og har samme præcedens som operatoren `+`.
- En operator e_1 `->>` e_2 som indsætter elementet repræsenteret ved udtrykket e_1 i køen repræsenteret ved udtrykket e_2. Eksempelvis evaluerer `5 ->> q2` til køen `[5 -> 1 -> 4]`. Operatoren `->>` er venstre associativ og har samme præcedens som operatoren $\star$.
- En operator `<<-` e, som returnerer næste element fra køen repræsenteret ved udtrykket e. Eksempelvis evaluerer `<<- [1 -> 5]` til 5. Operatoren `<<-` er ikke associativ og har højere præcedens end $\star$.

1. Du skal udvide lexer **FunLex.fsl** og parser **FunPar.fsy** med support for køer og de tre operatorer `++`, `<<-` og `->>`. Du skal udvide den abstrakte syntaks i **Absyn.fs** med `Queue` der repræsenterer et kø–udtryk og `Prim1`, der repræsenterer en operator der kun tager et argument, dvs., `<<-`.

   ```
   type expr =
     | CstI of int
     | CstB of bool
     | Queue of expr list     (* Exam 2023 *)
     | Prim1 of string * expr (* Exam 2023 *)
     ...
   ```

   Du ser den abstrakte syntaks for ovenstående eksempel `ex01` nedenfor:

   ```
   > open ParseAndRunHigher;;
   > fromString @"let q1 = [1 -> 2 -> 3] in
      let q2 = [1 -> 4] in
        let q3 = 5 ->> q1 ++ q2 in
          <<- q3
        end
      end
    end";;
   val it : Absyn.expr =
     Let
       ("q1", Queue [CstI 1; CstI 2; CstI 3],
        Let
          ("q2", Queue [CstI 1; CstI 4],
           Let
             ("q3", Prim ("++", Prim ("->>", CstI 5, Var "q1"), Var "q2"),
              Prim1 ("<<-", Var "q3"))))
   ```

Vis dine tilføjelser og at din løsning giver tilsvarende resultat for `ex01`. Lav yderligere 3 relevante test eksempler, og vis resulterende abstrakt syntaks. Forklar hvad dine 3 test eksempler tester for.

Hint: For at parse en kø $[e_n\texttt{->}\ldots\texttt{->}e_1]$ skal du parse et antal udtryk e_i, $1 \leq i \leq n$. Du finder inspiration til dette i parseren for parametre i micro–C, se non–terminal `Paramdecs` i parser specifikation **CPar.fsy** for micro–C.

2. Figur 4.3 på side 65 i PLC viser evalueringsregler for micro–ML, som vi har udvidet med køer. Nedenfor ses evalueringsregler for de nye konstruktioner:

$$(\textit{queue})\frac{\rho \vdash e_i \Rightarrow v_i,\ 1 \leq i \leq n}{\rho \vdash [e_n\texttt{->}\ldots\texttt{->}e_1] \Rightarrow \texttt{QueueV}[v_n,\ldots,v_1]}$$

$$(\texttt{++})\frac{\rho \vdash e_1 \Rightarrow \texttt{QueueV}[v_n,\ldots,v_1] \quad \rho \vdash e_2 \Rightarrow \texttt{QueueV}[w_m,\ldots,w_1]}{\rho \vdash e_1\texttt{++}e_2 \Rightarrow \texttt{QueueV}[v_n,\ldots,v_1,w_m,\ldots,w_1]}$$

$$(\texttt{->>})\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow \texttt{QueueV}[w_n,\ldots,w_1]}{\rho \vdash e_1\texttt{->>}e_2 \Rightarrow \texttt{QueueV}[v_1,w_n,\ldots,w_1]}$$

$$(\texttt{<<-})\frac{\rho \vdash e \Rightarrow \texttt{QueueV}[v_n,\ldots,v_1]}{\rho \vdash \texttt{<<-}\,e \Rightarrow v_1}$$

Reglen *queue* genererer en kø som repræsenteres ved $\texttt{QueueV}[v_n,\ldots,v_1]$. Udtryk evalueres fra venstre, e_n, mod højre, e_1. Når køer sættes sammen, kommer elementerne i e_1 efter elementerne i e_2 i køen, regel ++. I regel ->> indsættes element v_1 sidst i køen e_2. I regel <<- tages det forreste element i køen e ud, dvs. v_1.

Udvid typen `value` og funktionen `eval` i **HigherFun.fs**, således at udtryk med køer kan evalueres som defineret af reglerne ovenfor. Vi repræsenterer køer, `QueueV`, med den indbyggede `list`–type i F#.

```
type value =
  | Int of int
  | Closure of string * string * expr * value env (* (f, x, fBody, fDeclEnv) *)
  | QueueV of value list (* Exam 2023 *)
```

Vis dine tilføjelser og resultatet af at evaluere `ex01` samt dine tre test eksempler fra opgave 1 ovenfor. Eksempelvis giver `ex01`:

```
> open ParseAndRunHigher;;
> run (fromString @"let q1 = [1 -> 2 -> 3] in
   let q2 = [1 -> 4] in
     let q3 = 5 ->> q1 ++ q2 in
       <<- q3
     end
   end
 end");;
val it : HigherFun.value = Int 4
```

Ovenstående passer med at det forreste tal i `q2` er 4.

Opgave 4 (20%) Micro–C: Records

I denne opgave udvider vi sproget micro–C, som beskrevet i kapitel 7 og 8 i PLC, med *records*. Koden der anvendes her findes i kataloget `Lectures/Lec06/MicroC` i kursets git repository.
I eksemplet **record.c** nedenfor defineres en record `coord` i funktionen `main`. Recorden `coord` indeholder 3 felter: `x` og `y` af type `int` samt en tabel `a` med 5 elementer af type `int`. Felterne i en record tilgås med *dot*–syntaks, f.eks. `coord.x`. I eksemplet nedenfor erklæres også en lokal variabel `x`, som ikke skal forveksles med `coord.x`.

```
void main() {
  record coord = {
    int x,
    int y,
    int a[5]
  };

  int x;

  coord.x = 4;
  coord.y = 7;
  coord.a[0] = 6;
  x = 9;
  print (coord.x + coord.y + coord.a[0] + x);
}
```

Opgaven er at oversætte ovenstående program, som du gemmer i filen **record.c**, og køre programmet således at 26 printes.

```
> open ParseAndComp;;
> compile "record";;
val it : Machine.instr list =
  [LDARGS; CALL (0, "L1"); STOP; ...; RET -1]


% java Machine record.out
26
Ran 0.02 seconds
```

For at implementere records udvider vi typen `typ` i filen **Absyn.fs** med en ny record type `TypR`:

```
type typ =
  | TypI                          (* Type int                    *)
  ...
  | TypR of (typ * string) list (* Record type, exam 2023      *)
```

Record typen `TypR` tager en liste af par med type og navn på felterne i recorden. Med denne udvidelse, samt udvidelse af lexer og parser, kan vi udtrykke programmet **record.c** med følgende abstrakte syntaks.

```
> fromFile "record.c";;
val it : Absyn.program =
Prog
 [Fundec
  (None, "main", [],
   Block [Dec (TypR [(TypI, "x"); (TypI, "y"); (TypA (TypI, Some 5), "a")], "coord");
          Dec (TypI, "x");
         Stmt (Expr (Assign (AccVar "coord.x", CstI 4)));
         Stmt (Expr (Assign (AccVar "coord.y", CstI 7)));
         Stmt (Expr (Assign (AccIndex (AccVar "coord.a", CstI 0), CstI 6)));
         Stmt (Expr (Assign (AccVar "x", CstI 9)));
         Stmt (Expr (Prim1 ("printi",
         Prim2("+", Prim2("+", Prim2("+", Access (AccVar "coord.x"),
```

```
                                        Access (AccVar "coord.y")),
                                Access (AccIndex (AccVar "coord.a", CstI 0))),
                        Access (AccVar "x")))))])]
>
```

1. Udvid lexer specifikationen **CLex.fsl** og parserspecifikationen **CPar.fsy** med support for records, således at ovenstående abstrakte syntaks er resultatet af at parse programmet **record.c** med `fromFile "record.c";;`. Vis resultatet af at parse **record.c**.

   Hint: Du har brug for to nye tokens til a repræsentere punktum samt nøgleordet (eng. *keyword*) `record`.

   Hint: Du kan indsætte grammatik regel for at parse `record` *name* = `{` *vardec*, . . ., *vardec* `}` under non–terminalen `Vardec`. Du finder inspiration til at parse flere `Vardec`'s, ved at kigge på hvordan parametere til funktioner parses, se non–terminal `Paramdecs`.

   Hint: Du kan indsætte en grammatikregel under non–terminalen `Access` for at parse *recordName* `.` *fieldName*. Bemærk, at vi i den abstrakte syntaks repræsenterer `coord.x` ved `(AccVar "coord.x")`. I og med at det ikke er lovligt at definere variabelnavne der inkluderer et punktum i navnet, vil det sammensatte navn ikke blive blandet sammen med variabelnavne der ikke er records.

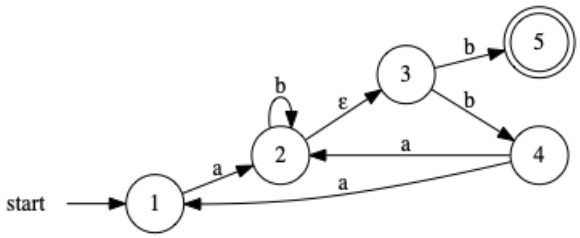
2. Forklar, ved at henvise til relevant lexer og parser specifikation eller F# kode, hvorfor vi kan konkludere at punktum ikke vil indgå i variabelnavne i micro–C.

3. Implementer allokering af records i oversætteren **Comp.fs**. Vis resultatet af at oversætte og køre **record.c**.

   Hint: Det er kun nødvendigt at udvide funktionen `allocate` med allokering af records for at kunne oversætte og køre programmet. Bemærk, at en record i sig selv ikke fylder på stakken, men kun felterne. I og med at vi sammensætter feltnavne, `"coord.x"` kan vi allokere disse felter på præcis samme måde som andre variabel erkæringer. Du kan derfor med fordel løbe over listen af felter og kalde `allocate` rekursivt.

Opgave 5 (10%) Regulære udtryk og automater

Betragt den ikke–deterministiske endelige automat (eng. *nondeterministic finite automaton*, NFA) nedenfor. Det anvendte alfabet er $\{\mathtt{a}, \mathtt{b}\}$. Der er i alt 5 tilstande, hvor tilstand 5 er den eneste accepttilstand.



1. Angiv alle årsager til at automaten er ikke–deterministisk.
2. Giv tre eksempler på strenge der genkendes af automaten.
3. Konstruer og tegn en deterministisk endelig automat (eng. *deterministic finite automaton*, DFA) der svarer til automaten ovenfor. Husk at angive starttilstand og accepttilstand(e). Du skal bruge en systematisk konstruktion svarende til den i forelæsningen eller som i Introduction to Compiler Design (ICD), eller Basics of Compiler Design (BCD).

   Hint: Du kan tage udgangspunkt i tabellen nedenfor.

DFA state	**move(a)**	**move(b)**	**NFA states**
S_0	...	...	$S_0 = \{1\}$
...	...	...	...