BSWU BPRD **IT University, E2019**

Skriftlig eksamen, Programmer som Data
December 2019

Version 1.01 af December 17, 2019

Dette eksamenssæt har 10 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside mandag den 16. december 2019 kl 08:00.
Besvarelsen skal afleveres elektronisk i LearnIt senest **tirsdag den 17. december 2019 kl 14:00** som følger:

- Besvarelsen skal uploades på kursets hjemmeside i LearnIt under **Submit Exam Assignment**.
- Der kan uploades en fil, som skal have en af følgende typer: `.txt`, `.pdf` eller `.doc`. 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.

Snydtjek

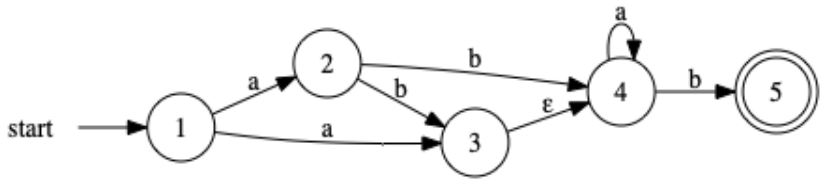
Til denne eksamen anvendes *Snydtjek*. Cirka 20% vil blive udtrukket af studeadministrationen i løbet af eksamen. Navne bliver offentliggjort på kursets hjemmeside tirsdag den 17. december klokken 14:00. Disse personer skal stille i et lokale der også annonceres på kursussiden tirsdag den 17. december klokken 15:00, dvs. kort tid efter deadline for aflevering i learnIt.

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%): Regulære udtryk og automater

Betragt den ikke–deterministiske endelige automat ("nondeterministic finite automaton", NFA) nedenfor. Det anvendte alfabet er $\{\texttt{a}, \texttt{b}\}$. 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. Giv en uformel beskrivelse af sproget (mængden af alle strenge) der beskrives af automaten.
4. Konstruer og tegn en deterministisk endelig automat ("deterministic finite automaton", DFA) der svarer til automaten ovenfor. Husk at angive starttilstand og accepttilstand(e). Du skal enten bruge en systematisk konstruktion svarende til den i forelæsningen eller som i Introduction to Compiler Design (ICD), eller Basics of Compiler Design (BCD), eller forklare hvorfor den resulterende automat er korrekt.
5. Angiv et regulært udtryk der beskriver mængden af strenge over alfabetet $\{\texttt{a}, \texttt{b}\}$, som beskrives af automaten ovenfor. Check og forklar at det regulære udtryk også beskriver mængden af strenge for din DFA.

Opgave 2 (25%) Micro–ML: Intervalcheck

Kapitel 5 i *Programming Language Concepts* (PLC) introducerer evaluering af et højereordens funktionssprog og kapitel 6 introducerer polymorf typeinferens.

Opgaven er at udvide funktionssproget med *intervalcheck*, som er udtryk af formen e `within [`e_1`,`e_2`]` der returnerer sand (1) eller falsk (0) afhængig af om $v_1 \leq v \leq v_2$ er opfyldt, hvor v, v_1 og v_2 er resultaterne af at evaluere e, e_1 og e_2.

I den abstrakte syntaks repræsenteres et intervalcheck med `InCheck` (e, e_1, e_2), hvor e, e_1 og e_2 er vilkårlige udtryk.

Betragt følgende micro–ML program

```
let x = 23 in x within [2+3,40] end
```

1. Udvid typen `expr` i `Absyn.fs` med `InCheck`, således at et intervalcheck kan repræsenteres.

   Vis de modifikationer du har lavet til `Absyn.fs` og at du kan repræsentere ovenstående micro–ML program som nedenfor.

   ```
   > open Absyn;;
   > InCheck(Var "x", Prim("+",CstI 2, CstI 3), CstI 40);;
   val it : expr = InCheck (Var "x",Prim ("+",CstI 2,CstI 3),CstI 40)
   ```

2. Udvid lexer `FunLex.fsl` og parser `FunPar.fsy` således at du kan parse intervalcheck, eksempelvis

   ```
   > open ParseAndRunHigher;;
   > fromString @"let x = 23 in x within [2+3,40] end";;
   val it : Absyn.expr =
     Let ("x",CstI 23,InCheck (Var "x",Prim ("+",CstI 2,CstI 3),CstI 40))
   ```

   Vis dine modifikationer til `FunLex.fsl` og `FunPar.fsy`. Lav 3 andre micro–ML programmer der anvender intervalcheck og vis den abstrakte syntaks som ovenfor.

3. Udvid funktionen `eval` i `HigherFun.fs`, med evaluering af `InCheck(`e,e_1,e_2`)`.

   ```
   > open ParseAndRunHigher;;
   > run (fromString @"let x = 23 in x within [2+3,40] end");;
   val it : HigherFun.value = Int 1
   ```

   Vis dine modifikationer til `HigherFun.fs` og resultatet af at evaluere programmerne fra forrige opgave.

4. Figur 6.1 på side 102 i PLC viser typeinferensregler for funktionssproget vi har udvidet med intervalcheck. Intervalcheck types med nedenstående typeregel.

$$(\textit{inCheck})\ \frac{\rho \vdash e : \texttt{int} \quad \rho \vdash e_1 : \texttt{int} \quad \rho \vdash e_2 : \texttt{int}}{\rho \vdash e\ \texttt{within}\ [e_1,e_2] : \texttt{bool}}$$

   Angiv et typeinferenstræ for udtrykket `let x = 23 in x within [2+3,40] end`. Du finder to eksempler på typeinferenstræer i figur 4.8 og 4.9 på side 72 i PLC.

Opgave 3 (25%) Micro-C: Print Variabel Environment

Kapitel 8 i *Programming Language Concepts* (PLC) introducerer sproget micro–C.

I denne opgave udvider vi oversætteren `Comp.fs` til under oversættelse at kunne udskrive indholdet af variabel environment `varEnv`. Målet er at lave en hjælpefunktion `ppVarEnv` *ve* af type `varEnv` → `string list`, som kan anvendes de steder i oversætteren `Comp.fs`, hvor man ønsker environment udskrevet, eksempelvis for at debugge et problem. For lokale variable returneres en streng formatteret efter følgende skabelon: `\n␣␣`*n*`:`*t*`␣at␣bp[`*o*`]`, hvor *n* er variabelnavn, *t* er variablens type og *o* er offset i forhold til base pointer `bp` i stack frame. For globale variable anvendes samme skabelon `\n␣␣`*n*`:`*t*`␣at␣Global[`*o*`]`, hvor `Global[`*o*`]` angiver den globale variabels absolutte adresse *o* i bunden af stakken.

Nedenfor ses et eksempel, som vi antager ligger i filen `exam1.c`.

```
// micro-C December 2019 example 1
int g;
void main() {
  int n;
  int *p;
  int *pn[1];
  int a[2];
  // exVarEnv svarer til varEnv her.
}
```

Programmet erklærer en global variabel `g` og 4 lokale variable `n`, `p`, `pn` og `a`. Nedenstående værdi `exVarEnv` svarer til indholdet af variabel environmet `varEnv` på det tidspunkt, hvor den sidste variabel `a` er erklæret.

```
let exVarEnv : varEnv =
  ([("a", (Locvar 6, TypA (TypI,Some 2)));
    ("pn", (Locvar 3, TypA (TypP TypI,Some 1)));
    ("p", (Locvar 1, TypP TypI));
    ("n", (Locvar 0, TypI));
    ("g", (Glovar 0, TypI))], 7)
```

Typen `varEnv` findes i `Comp.fs`.

1. Værdien `exVarEnv` af type `varEnv` i filen `Comp.fs` er et par (ve, o), hvor anden komponenten o har værdien 7. Forklar hvad o anvendes til og hvordan værdien 7 passer med værdien af første komponenten `[("a", (Locvar 6, TypA (TypI,Some 2))); ...]`.

2. For hver variabel skal vi formattere dens type. I filen `Absyn.fs` er en type defineret som

   ```
   type typ =
     | TypI                              (* Type int                    *)
     | TypC                              (* Type char                   *)
     | TypA of typ * int option          (* Array type                  *)
     | TypP of typ                       (* Pointer type                *)
   ```

   Implementer, i filen `Absyn.fs`, en funktion `ppTyp` *t* af type `typ -> string`, der returnerer en streng repræsentation af typen `t`. Du bestemmer selv den præcise formattering. Her er nogle eksempler til inspiration:

   ```
   > open Absyn;;
   > ppTyp TypI;;
   val it : string = "int"

   > ppTyp TypC;;
   val it : string = "char"

   > ppTyp (TypP TypI);;
   val it : string = "(*int)"

   > ppTyp (TypA(TypP TypI,Some 3));;
   val it : string = "((*int)[3])"
   ```

Du behøver ikke at følge præcis samme formattering, men du skal forklare hvordan du har valgt at formattere de forskellige typer.

Vis dine modifikationer og lav 3 nye test eksempler, som tester relevante egenskaber af din valgte formattering, i funktionen `ppTyp`. For hver test forklar hvilken egenskab der testes for.

3. I filen `Comp.fs` repræsenterer typen `var` adresser for henholdsvis globale og lokale variable:

```
type var =
     | Glovar of int         (* absolute address in stack           *)
     | Locvar of int         (* address relative to bottom of frame *)
```

Implementer, i filen `Comp.fs`, en funktion `ppVar` v af type `var -> string`, der returnerer en streng repræsentation af variablen's adresse på stakken. For globale variable er det en fast adresse og for lokale variable et offset i forhold til base pointer. Nedenstående eksempler viser hvorledes formatteringen ønskes.

```
> open Comp;;
ppVar (Glovar 1);;
val it : string = "Global[1]"

> ppVar (Locvar 3);;
val it : string = "bp[3]"
```

Vis dine modifikationer og kør din funktion `ppVar` på to nye eksempler og vis uddata.

4. Implementer, i filen `Comp.fs`, en funktion `ppVarTyp` $(s,(v,t))$ af type `string*(var*typ)` $\rightarrow$ `string` som formatterer variablen s, dens adresse v og type t efter skabelonerne ovenfor. Nedenstående eksempler viser hvorledes formatteringen kan se ud afhængig af din valgte formattering i `ppTyp`.

```
> open Absyn;;
> open Comp;;
> ppVarTyp ("a",(Locvar 11, TypA (TypI, Some 3)));;
val it : string = "a:(int[3]) at bp[11]"

> ppVarTyp ("g",(Glovar 0, TypI));;
val it : string = "g:int at Global[0]"
```

Vis dine modifikationer og kør din funktion `ppVarTyp` på to nye eksempler og vis uddata.

5. Implementer funktionen `ppVarEnv` af type `varEnv` $\rightarrow$ `string list`, som anvender skabelonerne for lokale og globale variable ovenfor.

Vis dine modifikationer. Kør din implementation af funktionen `ppVarEnv` på `exVarEnv` og vis uddata, eksempelvis:

```
> open Absyn;;
> open Comp;;
> let exVarEnv : varEnv =
-   ([("a", (Locvar 6, TypA (TypI,Some 2)));
-     ("pn", (Locvar 3, TypA (TypP TypI,Some 1)));
-     ("p", (Locvar 1, TypP TypI));
-     ("n", (Locvar 0, TypI));
-     ("g", (Glovar 0, TypI))], 7);;
val exVarEnv : varEnv =
  ([("a", (Locvar 6, TypA (TypI,Some 2)));
    ("pn", (Locvar 3, TypA (TypP TypI,Some 1))); ("p", (Locvar 1, TypP TypI));
    ("n", (Locvar 0, TypI)); ("g", (Glovar 0, TypI))], 7)

> ppVarEnv exVarEnv;;
val it : string list =
  ["g:int at Global[0]"; "n:int at bp[0]"; "p:(*int) at bp[1]";
   "pn:((*int)[1]) at bp[3]"; "a:(int[2]) at bp[6]"]
```

6. Vi ønsker nu at oversætteren skal udskrive variabel environment på skærmen hver gang den når til sidste statement eller declaration i en blok. Nedenfor ses koden for at oversætte en blok i `Comp.fs`:

```
let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match stmt with
    | If(e, stmt1, stmt2) ->
    ...
    | Block stmts ->
      let rec loop stmts varEnv =
          match stmts with
          | []     -> (snd varEnv, [])
          | s1::sr ->
            let (varEnv1, code1) = cStmtOrDec s1 varEnv funEnv
            let (fdepthr, coder) = loop sr varEnv1
            (fdepthr, code1 @ coder)
      let (fdepthend, code) = loop stmts varEnv
      code @ [INCSP(snd varEnv - fdepthend)]
    | Return None -> ...
```

Din opgave er at finde det sted, hvor der er genereret kode for den sidste statement i funktionen `loop` og indsætte kode, som udskriver variabel environment. Du kan med fordel anvende følgende hjælpefunktion:

```
let printVarEnv varEnv =
  List.iter (printf "\n%s") (ppVarEnv varEnv);
  printf "\n"
```

Vis dine modifikationer og vis uddata når du oversætter `exam1.c`, eksempelvis:

```
> open ParseAndComp;;
> compile "exam1";;

g:int at Global[0]
n:int at bp[0]
p:(*int) at bp[1]
pn:((*int)[1]) at bp[3]
a:(int[2]) at bp[6]
val it : Machine.instr list =
  [INCSP 1; LDARGS; CALL (0,"L1"); STOP; Label "L1"; INCSP 1; INCSP 1; INCSP 1;
   GETSP; CSTI 0; SUB; INCSP 2; GETSP; CSTI 1; SUB; INCSP -7; RET -1]
```

Opgave 4 (30%) Micro–C: Print Current Stack Frame

Kapitel 8 i *Programming Language Concepts* (PLC) introducerer sproget micro–C. Derudover introduceres en micro–C bytekode maskine i Java (`Machine.java`).

I denne opgave tilføjer vi et nyt statement `printCurFrame`, som udskriver alle lokale variable i scope på det givne sted samt deres indhold på skærmen. Dette er en fortsættelse af opgave 3, men kræver ikke at opgave 3 er løst.

Betragt nedenstående program `exam2.c`, som er en udvidelse af `exam1.c` fra opgave 3.

```
// micro-C December 2019 example 2
int g;
void main() {
  g=42;
  int n;
  n=1234;
  int *p;
  p = &n;
  int *pn[1];
  pn[0] = &g;
  int a[2];
  a[0] = 2;
  a[1] = 4;
  printCurFrame;
}
```

Målet med opgaven er at få uddata svarende til nedenstående, når programmet afvikles med Java bytekode maskinen:

```
$ java Machine exam2.out
Current Stack Frame (bp=3):
  n at bp[0] = 1234
  p at bp[1] = 3
  pn at bp[3] = 5
  a at bp[6] = 7

Ran 0.019 seconds
```

Som det ses udskrives alle lokalt definerede variable i scope på det tidspunkt hvor `printCurFrame;` indsættes. For hver variabel udskrives navn, dens placering i stack frame i forhold til base pointer `bp` samt dens nuværende værdi. For variable af type array, `pn` og `a`, udskrives kun indholdet af cellen, svarende til variablens stakadresse. Eksempelvis for `a` udskrives indholdet af `bp[6]`, som er 7 i ovenstående eksempel og ikke hele arrayets indhold. Base pointer `bp` peger på stak adresse 3.

1. Antag at nedenstående er uddata når `exam2.out`, der er resultatet af at oversætte `exam2.c`, afvikles med trace:

   ```
   $ java Machinetrace exam2.out
   [ ]{0: INCSP 1}
   ...
   [ 42 6 -999 1234 3 0 5 2 4 7 ]{89: PRINTCURFRM}
   Current Stack Frame (bp=3):
     n at bp[0] = 1234
     p at bp[1] = 3
     pn at bp[3] = 5
     a at bp[6] = 7
   [ 42 6 -999 1234 3 0 5 2 4 7 ]{104: INCSP -7}
   [ 42 6 -999 ]{106: RET -1}
   [ 42 -999 ]{6: STOP}
   ```

   Beskriv indholdet af stakken ved instruktion `PRINTCURFRM` (programadresse 89) ved at udfylde nedenstående skema. Du kan antage at base pointer `bp` er lig 3.

```
Stack Addr | Value | Description
      0    |    42 |
      1    |     6 | Retur adresse fra main.
      2    |  -999 |
bp->  3    |  1234 |
      4    |     3 |
      5    |     0 |
      6    |     5 |
      7    |     2 |
      8    |     4 |
      9    |     7 |
```

Hint: Layout af stack frames er beskrevet i PLC kapitel 8. Layout af arrays er vist på slide 21 til lektion 6 (`lecture06.pdf`).

De efterfølgende opgaver er at implementere `printCurFrame`.

2. Du skal udvide lexer `CLex.fsl`, parser `CPar.fsy` og `Absyn.fs` med support for `printCurFrame;`. Du ser den abstrakte syntaks for ovenstående eksempel `exam2.c` nedenfor:

```
> open ParseAndComp;;
> fromFile "exam2.c";;
val it : Absyn.program =
  Prog
    [Vardec (TypI,"g");
     Fundec
       (None,"main",[],
        Block
          [Stmt (Expr (Assign (AccVar "g",CstI 42))); Dec (TypI,"n");
           Stmt (Expr (Assign (AccVar "n",CstI 1234))); Dec (TypP TypI,"p");
           Stmt (Expr (Assign (AccVar "p",Addr (AccVar "n"))));
           Dec (TypA (TypP TypI,Some 1),"pn");
           Stmt
             (Expr (Assign (AccIndex (AccVar "pn",CstI 0),Addr (AccVar "g"))));
           Dec (TypA (TypI,Some 2),"a");
           Stmt (Expr (Assign (AccIndex (AccVar "a",CstI 0),CstI 2)));
           Stmt (Expr (Assign (AccIndex (AccVar "a",CstI 1),CstI 4)));
           Stmt PrintCurFrame])]

>
```

Vis dine rettelser og at din løsning giver tilsvarende resultat for `exam2.c`.

3. For at udskrive en stack frame på køretid, implementerer vi en ny bytekode instruktion `PRINTCURFRM` i `Machine.fs`:

```
type instr =
  ...
  | PRINTCURFRM of (int * string) list (* [(o_1,s_1);...;(o_n,s_n)]. *)
```

`PRINTCURFRM` tager en liste af par $[(o_1, v_1); \ldots; (o_n, v_n)]$, hvor o_i er offset i stack frame i forhold til base pointer `bp` for variabel v_i, $1 \leq i \leq n$. Eksempelvis er `PRINTCURFRM` for environment `exVarEnv` fra opgave 3 vist nedenfor. Den globale variabel `g` er ikke med, da den ikke er en del af stack frame.

```
PRINTCURFRM [(0, "n"); (1, "p"); (3, "pn"); (6, "a")]
```

For at emitte `PRINTCURFRM`, funktion `emitints` i `Machine.fs`, anvender vi nedenstående indkodning:

	pc	$pc+1$	$pc+2$	$pc+3$	$pc+4$	$pc+\cdots$	$pc+3+l_1$	$\cdots$
...	PRINTCURFRM	n	o_1	l_1	$fChar_1$	$\cdots$	$lChar_1$	$\cdots$

$\cdots$	o_n	l_n	$fChar_n$	$\cdots$	$lChar_n$	$\cdots$

Af figuren fremgår at instruktionen `PRINTCURFRM` optager $2 + \sum_{i=1}^{n}(2 + v_i.\texttt{length})$ ord, hvor n er antallet af variable og $v_i.\texttt{length}$ er længden af variabelnavn v_i. Dette skal bl.a. benyttes i funktionen `makelabenv` i `Machine.fs`.

I funktionen `emitints` skal du konvertere strengene til karakterer og opbygge bytekoden som vist ovenfor. Du kan tage udgangspunkt i nedenstående skabelon.

```
let rec emitints getlab instr ints =
    match instr with
    | Label lab      -> ints
    ...
    | PRINTCURFRM env   ->
        let codeString (s:string) = s.Length :: [for c in s -> (int) c]
        let codeVar (i,s) C = i :: (codeString s @ C)
        CODEPRINTCURFRM ::
        ...
        (List.foldBack ... env ints)
```

Vis og forklar alle dine rettelser i filen `Machine.fs`.

4. Bytekode maskinen `Machine.java` skal udvides med `PRINTCURFRM`.

	Instruction	**Stack before**		**Stack after**	**Effect**
0	`CSTI` i	s	$\Rightarrow$	s, i	Push constant i
...					
26	`PRINTCURFRM`, n, $o_1, l_1, fChar_1, \cdots, lChar_1$, ... $o_n, l_n, fChar_n, \cdots, lChar_n$	s	$\Rightarrow$	s	Prints current stack frame on the console and continues execution at the instruction following the last character $lChar_n$. No changes to the stack.

Du kan anvende følgende skabelon for at implementere `PRINTCURFRM` i funktionen `execcode` i `Machine.java`

```
static int execcode(int[] p, int[] s, int[] iargs, boolean trace) {
 int bp = -999; // Base pointer, for local variable access
 int sp = -1; // Stack top pointer
 int pc = 0; // Program counter: next instruction
 for (;;) {
   if (trace)
     printsppc(s, bp, sp, p, pc);
   switch (p[pc++]) {
   ...
   case PRINTCURFRM:
       System.out.print("Current Stack Frame (bp=" + bp + "):\n");
       int N = ... // Number of variables.
       for (int i=0; i<N;i++) {
         int o = ...   // Offset of variable v_i.
         System.out.print("  ");
         printStr(p,pc);  // Print variable v_i.
         pc += ...   // Progress pc to next variable.
         System.out.print(" at bp[" + o + "] = "+s[bp+o]);
         System.out.print("\n");
       }
       break;
   default:
     throw new RuntimeException("Illegal instruction " + p[pc-1]
```

```
                                        + " at address " + (pc-1));
    }
  }
}
```

Nedenstående funktion `printStr` udskriver karakterne $fChar_i$ til $lChar_i$ forudsat at `pc` peger på l_i, $1 \leq i \leq n$.

```
static void printStr(int[] p, int pc) {
  for (int i=0; i<p[pc]; i++)
    System.out.print((char)(p[pc+1+i]));
}
```

Vis og forklar alle rettelser i `Machine.java`.

5. For at implementere oversættelsen af `PrintCurFrame` i `Comp.fs` kan du anvende nedenstående skabelon:

```
let rec cStmt stmt (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match stmt with
    | ...
    | PrintCurFrame ->
        let varEnv' = List.filter (fun (_,(v,_)) -> isLocvar v) (fst varEnv)
        [PRINTCURFRM (List.map (fun (s,(v,_)) -> ...)
                               (List.rev varEnv'))]
    | ...
```

De to hjælpefunktioner `isLocvar` og `getOffset` er vist nedenfor:

```
let isLocvar = function
  | Glovar _ -> false
  | Locvar _ -> true

let getOffset = function
  | Glovar i -> i
  | Locvar i -> i
```

Vis og forklar implementationen af `PrintCurFrame` og specielt hvorfor det er nødvendigt at filtrere med `isLocvar`.

Vis at du kan oversætte og køre programmet `exam2.c`.