**BSWU BPRD** **IT University, E2018**

# Skriftlig eksamen, Programmer som Data
# 3.–4. januar 2019

Version 1.0 af 2018-12-01

Dette eksamenssæt har 8 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside onsdag 3. januar 2018 kl 08:00.
Besvarelsen skal afleveres elektronisk i LearnIt senest **torsdag 4. januar 2018 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 5 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 torsdag den 4. januar klokken 14:00. Disse personer skal stille i et lokale der også annonceres på kursussiden torsdag den 4. januar 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 (25 %): Icon

Kapitel 11 i *Programming Language Concepts* (PLC) introducerer "continuations" og "back tracking" samt sproget Icon.

I Icon kan vi f.eks. skrive `write(7 < (1 to 10))`. Ved at anvende implementationen i filen `Icon.fs` fra lektion 10 kan vi udtrykke dette i abstrakt syntaks:

```
let iconEx1 = Write(Prim("<",CstI 7,FromTo(1,10)))
```

og køre eksemplet, der udskriver tallet 8 med mere på skærmen, inde fra F# fortolkeren:

```
> run iconEx1;;
8 val it : value = Int 8
```

1. Omskriv eksemplet `iconEx1`, så værdierne 8 9 10 udskrives på skærmen, fx.:

   ```
   > run ...
   8 9 10 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Forklar hvorledes du får udskrevet alle 3 tal. Der gives flest point til løsninger, der bygger på eksemplet `iconEx1`.

2. Betragt nedenstående Icon udtryk

   ```
   let iconEx2 = Every(Write(And(FromTo(1,4),
                                 And(Write (CstS "\n"),FromTo(1,4)))))
   ```

   som udskriver nedenstående på skærmen når det afvikles.

   ```
   > run iconEx2;;

    1 2 3 4
    1 2 3 4
    1 2 3 4
    1 2 3 4 val it : value = Int 0
   ```

   Forklar hvorfor udtrykket giver ovenstående resultat.

   Omskriv eksemplet `iconEx2`, så nedenstående udskrives på skærmen, fx.:

   ```
   > run ...
    1 2 3 4
    2 4 6 8
    3 6 9 12
    4 8 12 16 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Der gives flest point til løsninger, der bygger på eksemplet `iconEx2`.

3. Udvid implementationen af Icon med en ny generator `Find(`*`pat`*`,`*`str`*`)`, som genererer en sekvens af indekser i strengen *str*, hvori mønsteret *pat* findes som en delstreng. `Find` starter med at søge efter *pat* fra starten af *str* og returnerer indekset for det første match den finder. Derefter starter den efter foregående match og returnerer indekset på næste match indtil den fejler, når der ikke længere findes et match. `Find` fejler, hvis *pat* eller *str* er de tomme strenge eller *pat* ikke findes som en delstreng i *str*. Første indeks i strengen *str* er 0 (som i F#).

   Nedenstående illustrerer brugen af `Find`.

   ```
   let str = "Hi there - if there are anyone"
   > val str : string = "Hi there - if there are anyone"
   run (Every(Write(Find("there",str))))
   > 3 14 val it : value = Int 0
   ```

Hint: .NET metoden `String.IndexOf` kan med fordel benyttes.

4. Lav 3 relevante testeksempler af `Find` ovenfor, og vis resultatet af at evaluere dem.

5. Skriv, og evaluer, et Icon udtryk, som udskriver indekser på alle tegn `"e"`, der findes i strengen `"Hi there - if there are anyone"`, fx.:

   ```
   > run ...
   5 7 16 18 22 29 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar.

   Omskriv dit svar til ovenstående, således at det kun er alle indekser på `"e"`, som er større end 10, der udskrives, fx.:

   ```
   > run ...
   16 18 22 29 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar.

# Opgave 2 (15 %): Parsing Records i micro–ML

Kapitel 5 i *Programming Language Concepts* (PLC) introducerer evaluering af et højereordens funktionssprog, micro–ML. Opgaven er at udvide funktionssproget med muligheden for at anvende simple records. Nedenfor ses 5 eksempler, som viser hvorledes records skabes og hvorledes felter i records tilgås.

```
let x = { } in x end (* ex1 *)

let x = {field1 = 32} in x.field1 end (* ex2 *)

let x = {field1 = 32; field2 = 33} in x end (* ex3 *)

let x = {field1 = 32; field2 = 33} in x.field1 end (* ex4 *)

let x = {field1 = 32; field2 = 33} in x.field1+x.field2 end (* ex5 *)
```

Opgaven er at udvide lexer (`FunLex.fsl`) og parser (`FunPar.fsy`) således at records svarende til eksemplerne ovenfor er understøttet i micro–ML. Grammatikken for records er vist nedenfor.

| Grammatik | | Kommentar |
|---|---|---|
| *Expr:* | $e$ .*field* | Adgang til felter. Udtrykket $e$ repræsenterer en record og *field* et felt i denne record. |
| | { $s_1 = e_1; \ldots; s_n = e_n$ } | Skabelse af en record. En record har 0, 1 eller flere felter $s_i$, som tildeles resultatet af udtrykkene $e_i$, $1 \leq i \leq n$. |

1. Filen `Absyn.fs` indeholder den abstrakte syntaks for micro–ML. Du skal udvide typen `expr`, med følgende to konstruktioner:

   ```
   type expr =
     ...
     | Field of expr * string
     | Record of (string * expr) list
     ...
   ```

   Adgang til felter udtrykkes med `Field(`$e$`,`$s$`)`, hvor $e$ repræsenterer udtrykket for en record og $s$ navnet på et felt i denne record. En record udtrykkes med `Record([(`$s_1, e_1$`);...;(`$s_n, e_n$`)])`, hvor $s_i$ er feltnavne og $e_i$ er udtryk, $1 \leq i \leq n$. Eksempelvis parses eksempel `ex2` ovenfor til den abstrakte syntaks nedenfor:

   ```
   Let ("x",Record [("field1", CstI 32)],Field (Var "x","field1"))
   ```

   Angiv den abstrakte syntaks, som værdier af typen `expr`, for de resterende eksempler, `ex1`, `ex3`, `ex4` og `ex5`, ovenfor.

2. Udvid lexer (`FunLex.fsl`) og parser (`FunPar.fsy`), således at records er understøttet med grammatikken vist ovenfor.

   Hint: Du har brug for nogle nye tokens, fx. `.` (punktum).

   Hint: Du har brug for at parse 0, 1 eller flere felter og udtryk, $s = e$, adskilt af semikolon. Parseren for micro–C har eksempler på at parse andre typer af sekvenser.

   Vis resultatet af at parse de 5 eksempler ovenfor og dokumenter, at din parser giver de forventede abstrakte syntakstræer.

# Opgave 3 (25 %): Evaluering af Records i micro–ML

I opgave 2 implementerer du en parser for understøttelse af records i micro–ML. I opgave 2 finder du udvidelse af den abstrakte syntaks (`Absyn.fs`) til repræsentation af records samt flere eksempler. Du behøver ikke at have løst opgave 2 for at løse denne opgave.

Evalueringsregler for micro-ML findes i figur 4.3 på side 65 i PLC. Nedenfor ses to regler der udvider micro-ML med evalueringsregler for records:

$$(e10)\ \frac{\rho \vdash e_i \Rightarrow v_i \quad 1 \le i \le n}{\rho \vdash \{s_1{=}e_1;\ldots;s_n{=}e_n\} \Rightarrow \{(s_1,v_1);\ldots;(s_n,v_n)\}}$$

$$(e11)\ \frac{\rho \vdash e \Rightarrow \{(s_1,v_1);\ldots;(s_n,v_n)\} \quad s = s_i \text{ for some } 1 \le i \le n}{\rho \vdash e.s \Rightarrow v_i}$$

Regel *e10* skaber en ny record, hvor felterne $s_i$ er parret med resultatet af at evaluere udtrykkene $e_i$, dvs. $v_i$. Notationen $\{(s_1\ ,\ v_1);\ \ldots;\ (s_n\ ,\ v_n)\}$ benyttes til at repræsentere værdien af en record. Regel *e11* anvendes til at tilgå et felt i en record og dermed få værdien, som feltet er bundet til. Nedenfor vises evaluering af at tilgå et felt i en record med to felter.

$$(e11)\ \frac{(e10)\ \dfrac{(e1)\ \dfrac{}{\rho \vdash 32 \Rightarrow 32} \quad (e1)\ \dfrac{}{\rho \vdash 3 \Rightarrow 3}}{\rho \vdash \{\texttt{field1}{=}32;\texttt{field2}{=}3\} \Rightarrow \{(\texttt{field1},32);(\texttt{field2},3)\}}}{\rho \vdash \{\texttt{field1}{=}32;\texttt{field2}{=}3\}.\texttt{field2} \Rightarrow 3}$$

1. Tegn et evaluerings træ, med reglerne i figur 4.3 på side 65 i PLC, samt *e10* og *e11* ovenfor, for udtrykket nedenfor (`ex2` fra opgave 2).

   $$(\cdots)\ \frac{\cdots}{[] \vdash \texttt{let x=}\{\texttt{field1=}32\}\ \texttt{in x.field1 end} \Rightarrow \cdots}$$

   Husk at angive hvilke regler du gør brug af.

2. For at kunne repræsentere records, som værdier, udvider vi typen `value` i filen `HigherFun.fs` med en ny konstruktion `RecordV`:

   ```
   type value =
     | Int of int
     | RecordV of (string * value) list
     | Closure of string * string * expr * value env
   ```

   Konstruktionen `RecordV`[$(s_1, v_1)$;...;$(s_n,v_n)$] repræsenterer en record som værdi. Hvert felt $s_i$ er parret med dets værdi $v_i$. Eksempelvis giver resultatet af at evaluere udtrykket `Record[("field1",CstI 32);("field2",CstI 33)]` værdien `RecordV[("field1",Int 32);("field2",Int 33)]`.

   Udvid funktionen `eval` i `HigherFun.fs`, med evaluering af records svarende til *e10* og *e11* ovenfor. Reglerne beskriver ikke hvad der skal ske, hvis der er sammenfald i feltnavne. Du skal beslutte og dokumentere, hvad du vil gøre i dette tilfælde. Eksempelvis vil evaluering af eksempel `ex2` fra opgave 3 give følgende:

   ```
   > open Absyn;;
   > ParseAndRunHigher.run (Let ("x",Record [("field1", CstI 32)],
                                    Field (Var "x","field1")));;
   val it : HigherFun.value = Int 32
   ```

3. Lav mindst 5 eksempler med records, hvor du bl.a. tester din beslutning om sammenfaldne feltnavne. Vis resultatet af at køre eksemplerne og forklar hvorvidt resultatet er som forventet. Du vælger selv om du vil lave eksemplerne i konkret syntaks og anvende parseren i opgave 2 eller om du laver eksemplerne i abstrakt syntaks.

## Opgave 4 (20 %): Breakpoints i micro–C

Kapitel 8 i *Programming Language Concepts* (PLC) introducerer sproget micro–C. Derudover introduceres en micro–C bytekode maskine i Java (`Machine.java`) og i C (`machine.c`). Begge bytekode maskiner kan trace programafvikling. Eksempelvis får vi ved at køre eksempel `ex1.c` med `machine.c` og tracing følgende uddata (afviklet på Mac):

```
$ ./machine -trace ex1.out 3
[ ]{0:LDARGS}
[ 3 ]{1:CALL 1 5}
[ 4 -999 3 ]{5:GOTO 21}
[ 4 -999 3 ]{21:GETBP}
...
```

Hver linie repræsenterer stakken og udskrives efter afvikling af hver bytekode instruktion. Dette kan for større programmer producere meget information der er svært at finde rundt i samt sløve programafvikling væsentligt.

I denne opgave tilføjer vi et nyt statement `break` *e*, hvis formål er, at man kan indsætte breakpoints i koden og dermed kontrollere, hvornår man vil have vist stakken. For hvert breakpoint angiver man et udtryk *e*, som skal evaluere til enten sand eller falsk. Hvis sand skal bytekode maskinen vente på, at brugeren trykker på ENTER tasten inden programafvikling fortsætter. For at simplificere opgaven, udskriver vi stakken, som den ser ud efter evaluering af *e*, i samme format, som ved trace.

Nedenfor ses to eksempler på anvendelsen af `break` for eksempel `ex1.c`. Til venstre ventes ikke på at bruger trykker på ENTER tasten efter hvert breakpoint. Bytekode maskinen i C er anvendt på en Mac.

```
Kildekode:

void main(int n) {
  while (n > 0) {
    print n;
    break false;
    n = n - 1;
  }
  println;
}

Oversættelse (Mac):

$ mono microcc.exe ex1.c
Micro-C backwards compiler v ...
Compiling ex1.c to ex1.out

Eksekvering (Mac):

$ ./machine ex1.out 3
3 [ 4 -999 3 0 ]{15:IFZERO 18}
2 [ 4 -999 2 0 ]{15:IFZERO 18}
1 [ 4 -999 1 0 ]{15:IFZERO 18}

Used   0.000 cpu seconds
$
```

```
Kildekode:

void main(int n) {
  while (n > 0) {
    print n;
    break true;
    n = n - 1;
  }
  println;
}

Oversættelse (Mac):

$ mono microcc.exe ex1.c
Micro-C backwards compiler v ...
Compiling ex1.c to ex1.out

Eksekvering (Mac):

$ ./machine ex1.out 3
3 [ 4 -999 3 1 ]{15:IFZERO 18}
Press ENTER to Continue

2 [ 4 -999 2 1 ]{15:IFZERO 18}
Press ENTER to Continue

1 [ 4 -999 1 1 ]{15:IFZERO 18}
Press ENTER to Continue


Used   0.000 cpu seconds
$
```

Din opgave er at implementere `break` *e* som vist ovenfor, hvilket inkluderer:

- at du skal udvide lexer og parser med support for `break` *e*, hvor *e* kan være et vilkårligt udtryk understøttet af micro–C. Du kan antage, at *e* evaluerer til 0 (falsk) eller 1 (sand).

- at du skal udvide bytekode maskinen, `Machine.java` eller `machine.c` efter eget valg, med to nye bytekode instruktioner `BREAK` og `WAITKEYPRESS`:

| | Instruction | Stack before | | Stack after | Effect |
|---|---|---|---|---|---|
| 0 | `CSTI` $i$ | $s$ | $\Rightarrow$ | $s, i$ | Push constant $i$ |
| ... | | | | | |
| 26 | `BREAK` | $s$ | $\Rightarrow$ | $s$ | Prints current stack content on the console and continue execution. No changes to the stack. |
| 27 | `WAITKEYPRESS` | $s$ | $\Rightarrow$ | $s$ | Prints "*Press ENTER to Continue*" on the console and pause execution until the user press the ENTER key. No changes to the stack. |

  Hint: Du får brug for kode der kan afbryde programafvikling og afvente ENTER. For C kan du forsøge med `system("read");` for Mac og Linux og med `system("pause");` på Windows. For Java kan du forsøge med `try {System.in.read();} catch (Exception e) {...}`.

- at du skal lave et oversætterskema for oversættelse af `break` *e* til bytekode. Du kan se eksempler på oversætter skemaer i PLC figur 8.5 på side 151, eller slide 25 fra lektion 7 den 11. oktober.

- at du skal udvide oversætteren således at der genereres kode for `break` *e*. Du skal bl.a. udvide den abstrakte syntaks (`Absyn.fs`), tilføje bytekodeinstruktioner (`Machine.fs`), og generere kode. Det forventes at du benytter `Contcomp.fs` og ikke `Comp.fs`.

- at du laver mindst et eksempel med `break` *e*, hvor *e* ikke er en simpel konstant, `true` eller `false`. Eksempelvis kan man med et logisk udtryk såsom `(n%2 == 0)` styre at bruger kun skal trykke ENTER når `n` er lige. Variablen `n` kunne fx. være en tæller i en løkke.

## Opgave 5 (15 %): Arrays i micro–C

Betragt nedenstående micro–C program `exam01.c`.

```
void printArray (int a[]) {
  int i;
  i=0;
  /* Describe stack content at this point. */
  while (i<5) {
    print (a[i]);
    i=i+1;
  }
}

void main() {
  int a[5];
  int i;
  i=0;
  while (i<5) {
    a[i] = 42;
    i=i+1;
  }

  printArray(a);
  print 43;       /* Statement to be removed in question 2. */
}
```

Når programmet afvikles på en Mac fås nedenstående uddata:

```
$ ./machine exam01.out
42 42 42 42 42 43 Used   0.000 cpu seconds
$
```

1. Tegn og beskriv indholdet af stakken når programafviklingen når til det sted, hvor kommentaren er indsat i funktionen `printArray`. Du skal opdele stakken i aktiveringsposter (eng. *stack frames*) og angive basepeger (eng. *base pointer*) og stakpeger (eng. *stack pointer*).

   Hint: Du kan anvende den abstrakte maskines mulighed for at udskrive stakken under afvikling af programmet, f.eks. `java Machinetrace exam01.out` eller `./machine -trace exam01.out`. Du lavede noget lignende i opgave 8.1 i PLC.

2. Nedenfor ses uddata fra at afvikle programmet `exam01.c` med den sidste statement i `main` (`print 43;`) fjernet. Det antages at programmet er oversat med `Contcomp.fs` (og ikke `Comp.fs`).

   ```
   $ ./machine exam01.out
   2 1 4 3 1 Used   0.000 cpu seconds
   $
   ```

   Vi har blot fjernet `print 43;` og får nu et helt andet resultat, der ikke er som forventet.

   Forklar, hvorfor vi, efter at have fjernet `print 43;`, får ovenstående uddata.

   Hint: Hvis du anvender `Comp.fs` i stedet for `Contcomp.fs`, så får du det forventede resultat.