BSWU BPRD **IT University, E2017**

Skriftlig eksamen, Programmer som Data
3.–4. januar 2018

Version 1.0 af 2018-01-02

Dette eksamenssæt har 9 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside onsdag 3. januar 2018 kl 09:00.
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

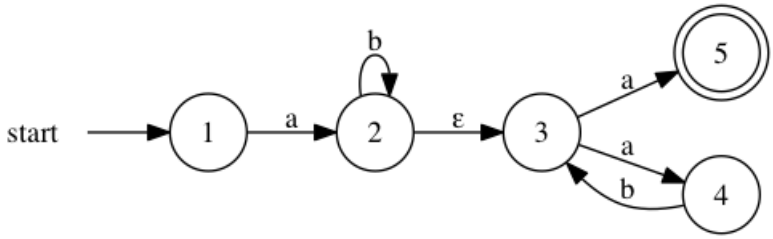
Til denne eksamen anvendes *Snydtjek*. Cirka 20% vil blive udtrukket af studeadministrationen i løbet af eksamen. Navne bliver offentliggjort på kursets hjemmeside torsdag den 4. januar klokken 14:00. Disse personer skal stille i et lokale der også annonceres på kursussiden torsdag den 4. januar klokken 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 (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 %): Icon

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

I Icon kan vi f.eks. skrive `every(write(1 to 2) | write(3 to 4))`. Ved at anvende implementationen i filen `Icon.fs` fra lektion 10 kan vi udtrykke dette i abstrakt syntaks:

```
let iconEx1 = Every(Write(Or(FromTo(1, 2), FromTo(3,4))))
```

og køre eksemplet, der udskriver tallet 1 med mere på skærmen, inde fra F# fortolkeren:

```
> run iconEx1;;
1 2 3 4 val it : value = Int 0
```

1. Omskriv eksemplet `iconEx1`, så værdierne 3 4 3 4 udskrives på skærmen, fx.:

   ```
   > run ...
   3 4 3 4 val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Forklar hvorledes du får udskrevet alle 4 tal. Der gives flest point til løsninger, der bygger på eksemplet `iconEx1`.

2. Skriv et Icon udtryk, som udskriver værdierne "I" "c" "o" "n" på skærmen:

   ```
   > run ...
   I c o n val it : value = Int 0
   ```

   hvor `...` repræsenterer dit svar. Du må ikke benytte `Bang` fra næste opgave.

3. Udvid implementationen af Icon med en ny generator `Bang(`*`str`*`)`, som genererer en sekvens af strenge der hver især er en karakter fra *str* i den rækkefølge de findes i *str*. `Bang` fejler, hvis *str* er den tomme streng.

   Eksempelvis giver nedenstående samme resultat som foregående opgave.

   ```
   > run (Every(Write(Bang "Icon")));;
   I c o n val it : value = Int 0
   ```

4. Udvid implementationen af Icon med en ny generator `BangN(`*`str`*`,`*`n`*`)`, som fungerer som `Bang(`*`str`*`)` ovenfor med den udvidelse at strengen *`str`* gentages *`n`* gange. Generatoren `Bang(`*`str`*`)` giver identisk uddata som `BangN(`*`str`*`, 1)`.

   ```
   > run (Every(Write(BangN("Icon", 1))));;
   I c o n val it : value = Int 0
   > run (Every(Write(BangN("Icon", 2))));;
   I c o n I c o n val it : value = Int 0
   ```

Opgave 3 (25 %): Enums i micro-ML

Kapitel 5 i *Programming Language Concepts* (PLC) introducerer evaluering af et højereordens funktionssprog. Opgaven er at udvide funktionssproget med muligheden for at anvende simple enumerations.

```
enum Weekend = Sat | Sun in
  Weekend.Sat
end
```

Variablen `Weekend` er en enumeration der kan antage to værdier `Sat` og `Sun`. Som vi kender fra enumerations i F# og C# tildeles værdierne et heltal startende fra 0, dvs. `Sat` har værdien 0 og `Sun` værdien 1. Eksemplet ovenfor evaluerer derfor til værdien 0.

Evalueringsregler for micro-ML findes på figur 4.3 side 63 i PLC. Nedenfor ses to regler der udvider micro-ML med enumerations.

$$(e10)\ \frac{\rho[x \rightarrow \texttt{Enum}\{E_0, \ldots, E_N\}] \vdash e \Rightarrow v}{\rho \vdash \texttt{enum}\ x = E_0 | \ldots | E_N\ \texttt{in}\ e\ \texttt{end} \Rightarrow v}$$

$$(e11)\ \frac{\rho(x) = \texttt{Enum}\{E_0, \ldots, E_N\} \qquad v = E_i \text{ for some } 0 \leq i \leq N}{\rho \vdash x.v \Rightarrow i}$$

Regel *e10* skaber en ny enumeration navngivet x med værdien $\texttt{Enum}\{E_0, \ldots, E_N\}$. I eksemplet ovenfor svarer E_0 til `Sat` og E_1 til `Sun`. Dvs. værdien bundet til variablen `Weekend` er $\texttt{Enum}\{\texttt{Sat}_0, \texttt{Sun}_1\}$, hvor værdierne 0 og 1 er angivet med subscript. Med regel $e11$ kan vi tilgå de mulige værdier med dot-notationen. Dvs. `Weekend.Sat` evaluerer til værdien 0 og `Weekend.Sun` til 1. Nedenfor vises evaluering af eksemplet ovenfor.

$$(e10)\ \frac{(e11)\ \dfrac{\rho'(\texttt{Weekend}) = \texttt{Enum}\{\texttt{Sat}_0, \texttt{Sun}_1\} \qquad \texttt{Sat} = \texttt{Sat}_i \quad i = 0}{\rho' = \rho[\texttt{Weekend} \rightarrow \texttt{Enum}\{\texttt{Sat}_0, \texttt{Sun}_1\}] \vdash \texttt{Weekend.Sat} \Rightarrow 0}}{\rho \vdash \texttt{enum Weekend} = \texttt{Sat} \mid \texttt{Sun in Weekend.Sat end} \Rightarrow 0}$$

1. Tegn et evaluerings træ, med reglerne i figur 4.3 (PLC) samt *e10* og *e11* ovenfor, for udtrykket nedenfor.

$$(e10)\ \frac{\cdots}{[] \vdash \texttt{enum Weekend} = \texttt{Sat} \mid \texttt{Sun in let r} = 1 + \texttt{Weekend.Sun in r} + 1 \texttt{ end end} \Rightarrow \cdots}$$

2. Vi udvider den abstrakte syntaks (fil `Absyn.fs` fra lektion 3 og 4) med support for enumerations således

```
type expr =
  | CstI of int
  ...
  | Enum of string * string list * expr (* Exam *)
  | EnumVal of string * string          (* Exam *)
  ...
```

`Enum`(n,vs,e) repræsenterer en enumeration navngivet n med de mulige værdier vs som en liste af strenge samt udtrykket e hvori den kan bruges. `EnumVal`(n,v) repræsenterer udvælgelsen af den angivne værdi v for enumeration n. Med eksemplet øverst i opgaven vil den abstrakte syntaks være

```
Enum ("Weekend",["Sat"; "Sun"],EnumVal ("Weekend","Sat"))
```

Udvid typen `expr` med ovenstående. Lav også den abstrakte syntaks for udtrykket fra opgave 1 ovenfor:

```
enum Weekend = Sat | Sun in let r = 1 + Weekend.Sun in r + 1 end end
```

Hint: Udtrykket starter med `Enum`($\cdots$) (eller lignende).

3. Udvid lexer og parser, således at enumerations er understøttet med syntaksen `enum` *x* = E_0 `|` $\cdots$ `|` E_N `in` *e* `end` og *x.v*.

   Hint: Du har brug for et nyt nøgleord `enum` samt tokens for `|` (bar) og `.` (punktum).

   Hint: For at parse E_0 `|` $\cdots$ `|` E_N kan du med fordel se hvorledes parametre til en funktion parses i micro-C. Se reglerne `Paramdecs` og `Paramdecs1` i filen `CPar.fsy` (lektion 6). Verificer at din parser giver samme abstrakte syntaks som den du lavede i opgave 2 ovenfor.

   Vis også den abstrakte syntaks for nedenstående eksempler og forklar hvorfor de er som forventet:

   - `enum OneTwo = One | Two in OneTwo.One end`
   - `enum One = One in One.One end`
   - `enum None = in 42 end`

   At et program kan parses betyder ikke nødvendigvis at programmet giver mening. Forklar hvorvidt du mener ovenstående repræsenterer valide programmer.

4. Udvid funktionen `eval` i `HigherFun.fs`, med evaluering af enumerations.

   Konstruktionen `Enum`(n,vs,e) skal returnere en værdi med enumeration navn n samt de mulige værdier vs. Til dette udvider vi typen `value` med `EVal`(n,vs), hvor første komponent er enumeration navn n og anden komponent de mulige værdier vs:

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

   Eksempelvis giver resultatet af at evaluere `Enum ("Weekend",["Sat"; "Sun"],Var "Weekend")` værdien `EVal ("Weekend",["Sat"; "Sun"])`.

   Hint: Du skal udvide omgivelsen `env` svarende til `let`–konstruktionen og evaluere kroppen e.

   Konstruktionen `EnumVal`(n,v) skal returnere index i for v i enumeration n som angivet i regel *e11* ovenfor. I praksis vil n være bundet til en værdi `Eval`(n,vs) i omgivelsen og resultatet er indexet af v i listen vs. Hvis v ikke findes i vs skal evalueringen fejle.

   Eksempelvis giver resultatet af at evaluere

   ```
   Enum ("Weekend",["Sat"; "Sun"],EnumVal ("Weekend","Sat"))
   ```

   resultatet 0

Opgave 4 (20 %): Micro-C Genereret Bytekode

I denne opgave kigger vi nærmere på den bytekode, som genereres af micro–C oversætteren, som beskrevet i kapitel 12 i PLC og gennemgået i lektion 11.

1. Betragt følgende micro–C program `microcEx2.c`:

```
void main() {
  int i;
  i = 3;
  print f(i);
}

int f(int n) {
  int i;
  i = 42;
  /* Describe stack content at this point. */
  return n+i;
}
```

   Tegn og beskriv indholdet af stakken når programafviklingen når til det sted, hvor kommentaren er indsat i funktionen `f`. 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 microEx2.out`. Du lavede noget ligende i opgave 8.1 i PLC.

2. Micro–C oversætteren `Contcomp.fs` fra kapitel 12 i PLC genererer følgende bytekode for programmet ovenfor:

```
  LDARGS;         Load commandline arguments - there are none
  CALL (0,"L1")   Call Main with 0 arguments.
  STOP            Return from Main and end program.
Label "L1"        Label for Main
  INCSP 1         ...
  GETBP
  CSTI 3
  STI
  INCSP -1
  GETBP
  LDI
  CALL (1,"L2")
  PRINTI
  RET 1
Label "L2"
  INCSP 1
  GETBP
  CSTI 1
  ADD
  CSTI 42
  STI
  INCSP -1
  GETBP
  LDI
  GETBP
  CSTI 1
  ADD
  LDI
  ADD
  RET 2
```

Til højre for de første 3 instruktioner og label er der indsat en kommentar der forklarer dens formål i forhold til kildekoden i `microcEx2.c`. Du skal tilføje tilsvarende kommentarer til de resterende instruktioner.

Hint: Du lavede ligende i opgave 8.1 og 8.4 i PLC.

3. Betragt følgende micro–C program `microcEx1.c`:

```
void main() {
  print f(100000);
}

int f(int n) {
  int r;
  r = 0;
  while (n > 0) {
    r = r + (42*2-34*2+32+3);
    n = n-1;
  }
  return r;
}
```

Når jeg oversætter med oversætteren fra kapitel 12 i PLC (`Contcomp.fs`) og afvikler programmet på min computer med Java maskinen tager det i snit 0,061 sekunder. I `while`–løkken laver vi bl.a. en beregning der kun behandler konstanter `(42*2-34*2+32+3)`. Opgaven er at udvide oversætteren `Contcomp.fs` således at den type beregninger, hvor der kun indgår konstanter, beregnes på oversættertid og ikke på køretid. På den måde må vi forvente at ovenstående program afvikles hurtigere.

Udpluk af den abstrakte syntaks for beregning af `r = r + (42*2-34*2+32+3);` ses nedenfor:

```
Assign(AccVar "r",
       Prim2("+",Access (AccVar "r"),
             Prim2("+",Prim2("+",Prim2("-",Prim2("*",CstI 42,CstI 2),
                                            Prim2("*",CstI 34,CstI 2)),
                                CstI 32),
                   CstI 3)))
```

Ideen er at vi i forbindelse med kodegenerering forsøger at reducere den abstrakte syntaks ved at lave beregninger der kun inkluderer konstanter inden vi genererer kode. Eksempelvis kan ovenstående reduceres til følgende abstrakte syntaks:

```
Assign(AccVar "r",
       Prim2("+",Access (AccVar "r"),CstI 51))
```

Vi begrænser opgaven til kun at håndtere de fire regnearter +, -, * og /, dvs. kun kigger på oversættelsen af `Prim2`–konstruktionen. Du kan anvende følgende skabelon i `Contcomp.fs`:

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) (C : instr list) : instr list =
    match e with
    ...
    | Prim2(ope, e1, e2) ->
      let e1' = reducePrim2 e1  (* Exam *)
      let e2' = reducePrim2 e2  (* Exam *)
      cExpr e1' varEnv funEnv
        (cExpr e2' varEnv funEnv
           (match ope with
            | "*"   -> MUL  :: C
            | "+"   -> ADD  :: C
            | "-"   -> SUB  :: C
            | "/"   -> DIV  :: C
          ...
```

I `Prim2` er indsat to kald til en funktion `reducePrim2` som du skal lave der forsøger at reducere udtrykkene `e1` og `e2` inden der genereres kode. En mulig skabelon for `reducePrim2` er

```
let rec reducePrim2 e = (* Exam *)
  let e' =
    match e with
      Prim2(ope,e1,e2) ->
        let e1' = reducePrim2 e1
      ...
    | e -> e
  if e <> e' then printfn "ReducePrim2: Expression %A reduced to %A" e e'
  e'
```

Når jeg afvikler eksempel `microcEx1.c` med brug af `reducePrim2` tager det i snit 0,041 sekunder, dvs. en optimering på cirka 33%.

Opgave 5 (10 %): Hyppighed af Kørt Bytekode

Når man studerer kode genereret af en oversætter med henblik på at finde på nye optimeringer kan det være en hjælp at have viden om hvilke instruktioner som hyppigst anvendes. Dette kan gøres ved oversættelse, dvs. hvad er hyppigheden af de genererede instruktioner og det kan gøres på køretid, dvs. hvad er hyppigheden af de instruktioner som afvikles.

I denne opgave skal du udvide den abstrakte maskine til at udskrive hyppigheden af de afviklede instruktioner. Til dette kan du anvende en tabel indekseret efter bytekode instruktionernes nummer og tælle op hver gang en instruktion afvikles.

Løsningen kan eksempelvis udskrive nedenstående efter afvikling af `microcEx1.c` som eksempel:

```
Byte Code Statistics:
CSTI : 500005
ADD : 300002
SUB : 100000
MUL : 0
DIV : 0
MOD : 0
EQ : 0
LT : 100001
NOT : 0
...
```

1. Du skal nu udvide den abstrakte maskine til at opsamle og udskrive tilsvarende statistik efter kørsel af et program. Du bestemmer selv om du vil udvide maskinen implementeret i C (`machine.c`) eller Java (`Machine.java`).

2. Vis den fulde statistik af at køre `microcEx1.c` med din løsning.