BSWU BPRD **IT University, E2015**

Skriftlig eksamen, Programmer som Data
7.–8. januar 2016

Dette eksamenssæt har 7 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside torsdag 7. januar 2016 kl 09:00.
Besvarelsen skal afleveres elektronisk i LearnIt senest **fredag 8. januar 2016 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 den ønskede funktion så den har netop den type og giver det resultat som opgaven kræver.

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*. Det betyder at cirka 20% vil blive udtrukket af studeadministrationen i løbet af eksamen. Navne på disse personer vil blive offentliggjort på kursets hjemmeside fredag den 8. januar klokken 14:00. Disse personer skal stille i lokale 2A18 fredag den 8. 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.

Vi forventer hele processen kan gøres på lidt over 1 time.

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 %): Regulære udtryk og automater

Betragt dette regulære udtryk over alfabetet $\{k, l, v, h, s\}$:

$$k(l|v|h)+s$$

Ved antagelse, at

k	svarer til	*kør*
l	svarer til	*ligeud*
v	svarer til	*venstre*
h	svarer til	*højre*
s	svarer til	*slut*

så beskriver det regulære udtryk strenge, der symboliserer køreture, eksempelvis turen fra hjem til ITU.

1. Giv nogle eksempler på strenge der genkendes af dette regulære udtryk. Giv en uformel beskrivelse af sproget (mængden af alle strenge) der beskrives af dette regulære udtryk.

2. Konstruer og tegn en ikke-deterministisk endelig automat ("nondeterministic finite automaton", NFA) der svarer til det regulære udtryk. 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.

3. Konstruer og tegn en deterministisk endelig automat ("deterministic finite automaton", DFA) der svarer til det regulære udtryk. 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.

4. Betragt nedenstående uendelige mængde af strenge over alfabetet $\{l, v, h\}$. For at blive i analogien med køreture kan du antage at *l* svarer til *ligeud*, *v* for *venstre* og *h* for *højre*. For eksempel vil strengen `vll` svare til *venstre*, *ligeud*, *ligeud*.

1 iteration: `l`, `vl`, `hl`

2 iteration: `ll`, `lvl`, `lhl`, `vll`, `vlvl`, `vlhl`, `hll`, `hlvl`, `hlhl`

3 iteration: `lll`, `llvl`, `llhl`, `lvll`, `lvlvl`, `lvlhl`, ...

Det regulære udtryk består af en enkelt repeterende komponent, som kan iterere 1 eller flere gange. Ovenfor ses hvorledes strenge genkendes af henholdsvis 1, 2 og delvist 3 iterationer af den repeterende komponent. For eksempel er strengene `l`, `vl` og `hl` samtlige strenge, som fås med bidrag af en enkelt iteration.

Angiv et regulært udtryk der beskriver denne mængde af ikke tomme strenge.

Følgende eksempler genkendes *ikke* af det regulære udtryk: `lv`, `lh`, `lvv`, `lvh`, `lhv` og `lhh`.

Hint: Der er 3^n strenge når den repeterende komponent itererer n gange, fx 3 for $n = 1$ og 9 for $n = 2$ ovenfor.

Opgave 2 (25 %): Referencer i Funktionssprog

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 referencer, således at de kan evalueres. I den abstrakte syntaks repræsenteres en reference med `Ref` *e*, hvor e er et vilkårligt udtryk. Et udtryk e derefereres med `Deref` *e* og et referenceudtryk e_1 opdateres med værdien af e_2 ved `UpdRef`(e_1,e_2).

1. Udvid typen `expr` i `Absyn.fs` med `Ref`, `Deref` og `UpdRef`, således at

   - referencer kan skabes, for eksempel `Ref(CstI 1)`.
   - referencer kan derefereres, for eksempel `Deref(Ref(CstI 1))`.
   - referencer kan opdateres, for eksempel `UpdRef(Ref(CstI 1), CstI 2)`.

   Vis (i udklip) de modifikationer du har lavet til `Absyn.fs`

2. Udvid typen `value` i `HigherFun.fs` med følgende konstruktion således at referenceværdier kan håndteres:

   ```
   RefVal of value ref
   ```

3. Udvid funktionen `eval` i `HigherFun.fs`, med evaluering af `Ref`, `Deref` og `UpdRef`, således at de opfylder følgende:

`Ref` e: Udtrykket e evalueres til værdien v og en referenceværdi returneres: `RefVal (ref` v`)`.

`Deref` e: Udtrykket e evalueres til værdien v. Hvis v er en referenceværdi, dvs. på formen `RefVal` v', så returneres indholdet af v'. Evalueringen skal fejle, hvis v ikke er en referenceværdi.

`UpdRef(`e_1`,`e_2`)`: Udtrykket e_1 evalueres til værdien v_1. Hvis v_1 er en referenceværdi, `RefVal` v_1', evalueres e_2 til værdien v_2 og indholdet af v_1' opdateres til v_2. Resultatet v_2 returneres. Evalueringen skal fejle, hvis v_1 ikke er en referenceværdi.

   Hint: Du skal anvende F#'s support af referencer, dvs. `ref`, `!` og `:=`. Bemærk at vores implementation af `UpdRef` er anderledes end `:=` i F#, hvor værdien `unit` returneres.

   Følgende to eksempler illustrerer definitionerne ovenfor. Det første udtryk evaluerer til værdien `Int 2`.

   ```
   Let("x",Ref(CstI 1),
       If (Prim("=",Deref(Var "x"),
                    CstI 1),
           UpdRef(Var "x", CstI 2),
           CstI 42))
   ```

   Det andet udtryk evaluerer til værdien `Int 6`.

   ```
   Let ("x",Ref (CstI 2),
        Prim ("+",UpdRef (Var "x",CstI 3),
                  Deref (Var "x")))
   ```

   Vis (i udklip) de modifikationer du har lavet til `HigherFun.fs` og giv en skriftlig forklaring af modifikationerne på 5–10 linjer.

4. Erklær mindst fire eksempler på udtryk af type `expr` hvori `Ref`, `Deref` og `UpdRef` indgår. Eksemplerne skal inkludere mindst en reference hvis værdi opdateres. Vis resultatet af at evaluere eksemplerne med `eval`. Du kan benytte funktionen `run` i `ParseAndRunHigher.fs`.

5. Udvid lexer og parser, således at referencer er understøttet med samme syntaks, som vi kender fra F#. Det skal for eksempel være muligt at skrive `"ref 1"`, `"!(ref 1)"` og `"(ref 1) := 2"` svarende til eksemplerne ovenfor (pind 1). Vis (i udklip) de modifikationer du har lavet til `FunLex.fsl` og `FunPar.fsy` og giv en skriftlig forklaring af modifikationerne på 5–10 linjer.

6. Omskriv de to eksempler fra pind 3 og dine egne eksempler fra pind 4 ovenfor således at de giver samme resultat når lexer og parser anvendes inden evaluering. Vis resultatet af at evaluere eksemplerne.

7. Figur 6.1 på side 98 i PLC viser typeinferensregler for funktionssproget vi har udvidet med referencer. Til at beskrive typen af referenceudtryk introducerer vi en ny type t `ref`. Eksempelvis vil `ref 1` have typen `int ref`, præcis som i F#. Dette kan beskrives med følgende tre typeregler, hvor operatorerne `!` og `:=` beskrives af de to sidste:

$$(ref)\frac{\rho \vdash e : t}{\rho \vdash \texttt{ref}\ e : t\ \texttt{ref}} \qquad (!)\frac{\rho \vdash e : t\ \texttt{ref}}{\rho \vdash\ !e : t} \qquad (:=)\frac{\rho \vdash e_1 : t\ \texttt{ref} \quad \rho \vdash e_2 : t}{\rho \vdash e_1 := e_2 : t}$$

   Angiv et typeinferenstræ for udtrykket `let x = ref 2 in (x := 3) + !x end`. Du finder to eksempler på typeinferenstræer i figur 4.8 og 4.9 på side 70 i PLC.

Opgave 3 (25 %): Strenge i List-C

I denne opgave udvider vi sproget List-C, som beskrevet i afsnit 10.7 i PLC, med strenge allokeret på hoben (*eng.* heap). Filerne der anvendes findes i `listc.zip` fra lektion 10. Spildopsamling (*eng.* Garbage collection) behøver ikke at fungere for at løse denne opgave. Som beskrevet nederst side 127 i PLC er der allerede support for at parse strenge i `CLex.fsl` og `CPar.fsy`; søg efter `CSTSTRING` i de to filer.
Figuren nedenfor angiver hvordan en streng repræsenteres på hoben.

	p	$p+1$	$p+2$		$p+sizeW+1$	
...	*header*	`lenStr`	`str[0]`	...	`str[lenStr-1]` '\0'	...

Afsnit 10.7.3 og 10.7.4 i PLC beskriver indholdet af *header*, der bl.a. indeholder et tag og størrelsen af blokken som antal ord (*eng.* words). I C, som `listmachine.c` er skrevet i, er strenge altid nul-termineret, dvs. slutter med '\0'. Et tegn fylder 8 bit og et 32 bit ord har plads til 4 tegn. En streng *str* er dermed repræsenteret med en header, efterfulgt af længden af strengen (*lenStr*), efterfulgt af strengens *lenStr* tegn og afsluttet med '\0'. Som eksempel er strengen `"Hi There"` repræsenteret nedenfor.

	p	$p+1$	$p+2$	$p+3$	$p+4$	
...	*header*	8	`Hi T`	`here`	'\0'	...

I eksemplet har vi i alt brug for 9 tegn (de 8 plus '\0'), hvilket fylder 3 ord, dvs. *sizeW* = 3.
Opgaven er at udvide List-C således at konstante strenge kan oprettes. Vi genbruger typen `dynamic` for strenge, se eksempel nedenfor.

```
void main() {
  dynamic s1;
  dynamic s2;
  s1 = "Hi there";
  s2 = "Hi there again";
}
```

Nedenfor følger en opskrift på at udvide List-C med strenge.

`Absyn.fs`: Tilføj `CstS` til typen `expr` som repræsenterer den konstante streng i den abstrakte syntaks.

`CPar.fsy`: Tilføj en regel med `CSTSTRING` således at en knude i den abstrakte syntaks med `CstS` oprettes for den konstante streng, fx.

```
AtExprNotAccess:
    Const       { CstI $1               }
  | CSTSTRING   { CstS $1               }
  ...
```

`Machine.fs`: Tilføj en instruktion `CSTS` `of` `string`, som repræsenterer den konstante streng. Ideen er at lægge strengen som del af programkoden med følgende layout

	pc	$pc+1$	$pc+2$	$pc+3$	...	$pc+strLen+1$	
...	CODECSTS	*lenStr*	*str[0]*	*str[1]*	...	*str[lenStr-1]*	...

Hvert tegn i strengen fylder et ord i programkoden. I funktionen `makelabenv` skal antallet af ord som instruktionen fylder angives. Af ovenstående ses at det er *strLen*+2. I funktionen `emitints` skal koden genereres. Hjælpefunktionen `explode` omdanner hvert tegn i strengen til en integer.

```
let explode s = [for c in s -> int c]
```

Programkoden for `CSTS` kan derefter genereres således:

```
| CSTS s  -> CODECSTS :: (String.length s) :: ((explode s) @ ints)
```

`Comp.fs`: I funktionen `cExpr` oversættes `CstS` nemt til instruktionen oprettet i `Machine.fs`:

```
  | CstS s          -> [CSTS s]
```

`listmachine.c`: Vi lader en streng få tagget 1: `#define STRINGTAG 1`. Vi lader `CSTS` repræsentere instruktionen genereret ovenfor. Dette giver følgende case i funktionen `execcode`:

```
case CSTS: {
  int lenStr = p[pc++];
  int sizeStr = lenStr + 1; // Extra for zero terminating string, \0.
  int sizeW = (sizeStr % 4 == 0)?sizeStr/4:(sizeStr/4)+1; // 4 chars per word
  sizeW = sizeW + 1; // Extra for string length.
  word* strPtr = allocate(STRINGTAG, sizeW, s, sp);
  s[++sp] = (int)strPtr;
  strPtr[1] = lenStr;
  char* toPtr = (char*)(strPtr+2);
  for (int i=0; i<lenStr; i++)
    toPtr[i] = (char) p[pc++];
  toPtr[lenStr] = '\0'; // Zero terminate string!
  printf ("The string \"%s\" has now been allocated.\n", toPtr); /* Debug */
} break;
```

Bemærk, at vi udskriver strengen på skærmen som debug, således at vi kan se at strengen er oprettet.

1. Vis (i udklip) de modifikationer du har lavet til filerne `Absyn.fs`, `CPar.fsy`, `Comp.fs`, `Machine.fs` og `listmachine.c` for at få ovenstående opskrift til at virke. Giv en skriftlig forklaring af modifikationerne på 5–15 linjer.
2. Vis med eksempelprogrammet ovenfor at strenge kan oprettes. Giv en skriftlig forklaring af den abstrakte syntaks, som skabes når eksempelprogrammet oversættes.

Opgave 4 (25 %): Interval check i micro-C

Med *interval check* menes et boolsk udtryk på formen $z\ op_1\ x\ op_2\ y$, hvor x, y og z er af typen `int`. Udtrykket $z\ op_1\ x\ op_2\ y$ er sandt hvis og kun hvis $z\ op_1\ x$ og $x\ op_2\ y$ begge er sande. De to operatorer op_1 og op_2 kan være en af følgende nye boolske operatorer: **`.==`**, **`.!=`**, **`.>`**, **`.<`**, **`.>=`** og **`.<=`**. Operatoren `.==` svarer til den eksisterende oprator == i micro-C og således også for de andre. Operatorerne til interval check har fået tilføjet et punktum for at simplificere parseren.
Nedenstående eksempel

```
void main() {
  print 2 .< 3 .< 4;                      // 2 < 3 && 3 < 4
  print 3 .< 2 .== 2;                     // 3 < 2 && 2 == 2
  print 3 .> 2 .== 2;                     // 3 > 2 && 2 == 2
  print (3 .> 2 .== 2) == (3 .> 1 .== 1); // (3 > 2 && 2 == 2) == (3 > 1 && 1 == 1)
  print (3 .> 2 .== 2) == 1;              // (3 > 2 && 2 == 2) == 1
}
```

skriver følgende på skærmen når det køres.

```
1 0 1 1 1
```

Som eksempel kan den abstrakte syntaks for `2 .< 3 .< 4` se således ud:

```
Andalso (Prim2 ("<",CstI 2,CstI 3),Prim2 ("<",CstI 3,CstI 4)))))
```

Filerne der anvendes findes i `microc.zip` fra lektion 6.

1. Udvid micro-C med interval check, således at ovenstående eksempel giver samme resultat. Vis (i udklip) de modifikationer du har lavet til `CLex.fsl` og `CPar.fsy`. Giv en skriftlig forklaring af modifikationerne på 5 – 10 linjer.

   Hint: Det er ikke nødvendigt at ændre `Comp.fs` eller introducere nye bytekode instruktioner.

   Hint: *shift/reduce* konflikter i den generede parser accepteres sålænge at ovenstående program fungerer.

2. Udvid programmet ovenfor med mindst 5 yderligere interval check, der tester korrektheden af din løsning til foregående delspørgsmål. Forklar på 5 – 10 linjer, i hvilken grad resultaterne er som forventet.