BSWU BPRD **IT University, E2013**

Skriftlig eksamen, Programmer som Data
2.–3. januar 2014

Dette eksamenssæt har 5 sider. Tjek med det samme at du har alle siderne.
Eksamenssættet udleveres elektronisk fra kursets hjemmeside torsdag 2. januar 2014 kl 09:00.
Besvarelsen skal afleveres på papir senest **fredag 3. januar 2014 kl 14:00** som følger:

- Besvarelsen skal afleveres til studieadministrationen (fløj 3D) på IT-Universitetet.
- Som altid skal der bruges ITU-projektforside, se http://studyguide.itu.dk/en/SWU/Your-Programme/Forms
- Som altid skal der afleveres tre eksemplarer.

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.

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 underskrevne erklæring:

> **Jeg erklærer hermed at jeg selv har lavet hele denne eksamensbesvarelse uden hjælp fra andre.**
> (underskrift) (dato)

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.

Baggrund for opgavesættet

Opgaverne går ud på at udvide micro-C sproget på forskellige måder som angivet nedenfor. Micro-C sproget er beskrevet i *Programming Language Concepts* kapitel 7 og 8 samt i de tilsvarende forelæsninger i kursusuge 40–41. Som udgangspunkt for opgavebesvarelsen skal du bruge implementationen af micro-C fra microc.zip, der kan downloades fra kursushjemmesiden. Brug den almindelige "forlæns" oversætter i `Comp.fs`, ikke den "baglæns" i `ContComp.fs`.

Opgave 1 (25 %): For-to-løkker

Udvid MicroC med en for-to-løkke som de kendes fra BASIC eller Pascal, så man kan skrive eksempelvis:

```
for i=0 to 100 do
  sum = sum+i;
```

For at gøre dette skal du udvide lexer- og parser-specifikationerne `CLex.fsl` og `CPar.fsy`.

Du kan dernæst udvide den abstrakte syntaks i `Absyn.fs` ved at definere en ny konstruktor `ForTo` i `stmt` typen, og tilføje et passende gren i `cStmt` funktionen i oversætteren i `Comp.fs`.

Men det er nok nemmere simpelthen at lade parseren generere passende abstrakt syntaks uden at ændre i `Absyn.fs` eller `Comp.fs`. En for-to-løkke på den generelle form:

```
for x = e1 to e2 do
  stmt
```

er nemlig ækvivalent med en blok af denne form, indeholdende en while-løkke:

```
{
  int x;
  x = e1;
  while (x <= e2) {
    stmt
    x=x+1;
  }
}
```

Det er altså nok at lade den semantiske aktion `{ ... }` i parseren konstruere abstrakt syntaks der bygger på de eksisterende konstruktorer såsom `Block`, `While`, `Expr`, `Dec`, `Stmt` og så videre.

1. Vis, i udklip, de modifikationer du har lavet til `CLex.fsl` og `CPar.fsy` og eventuelt `Absyn.fs` og `Comp.fs`, og giv en skriftlig forklaring af modifikationerne på 10–20 linjer.

2. Lav et testeksempel der viser at din implementation af for-løkker fungerer korrekt. Vis testeksemplet og forklar hvilket resultat det skal give.

3. Forklar (fx med angivelse af F#-udtryk og kommandolinjeordrer) hvordan du har oversat og kørt testeksemplet, og vis det faktiske resultat af at køre det.

(Husk at et problem i `fsyacc` giver 16 shift/reduce fejlmeldinger på grund af `nonassoc`-direktivet for tokens `LT`, `GE`, `LE`, osv. i `CPar.fsy`. Disse fejlmeldinger kan du se bort fra).

Opgave 2 (25 %): Sammensatte tildelinger

Udvid micro-C med sammensatte tildelinger af formen `x += e` og `x -= e`. Du skal modificere lexer- og parser-specifikationerne, den abstrakte syntaks, og oversætterens `cExpr` funktion. Ligesom for almindelige tildelinger `x = e` skal venstresiden `x` være en lvalue, dvs. af abstrakt syntaks typen `access`.

Venstresiden `x` må kun evalueres én gang. For eksempel skal udførelsen af:

```
arr[i=i+1] += 100;
```

kun udregne `arr[i=i+1]` én gang, og dermed kun lægge 1 til variabel `i`, ikke 2.

1. Vis (i udklip) de modifikationer du har lavet til `CLex.fsl` og `CPar.fsy` og `Absyn.fs` og `Comp.fs`, og giv en skriftlig forklaring af modifikationerne på 20–40 linjer.
2. Angiv et testprogram der viser at `arr[i=i+1] += 100` virker som det skal: både at det relevante element af `arr` er blevet øget med 100, og at `i` er blevet øget med 1.
3. Vis resultatet fra en kørsel af testprogrammet.

Opgave 3 (25 %): Simple arrayindekstjek

Som forklaret i forelæsningen fra uge 40 (lecture06.pdf, slide 19), og som det fremgår af `Comp.fs`, så repræsenterer micro-C et `n`-element array `int arr[n]` ved hjælp af `n+1` stakpladser, således:

	q	$q+1$		$q+n-1$	a	
...	arr[0]	arr[1]	...	arr[n-1]	q	...

Her er `arr[0] ... arr[n-1]` arrayets elementer, der efterfølges af q, som er adressen på `arr[0]`. Denne repræsentation kan udnyttes til at lave et "fattigmands"-indekstjek i micro-C: Hvis a er adressen på stakpladsen der indeholder q, så er $(a - q) = \texttt{n}$. Når man udfører koden svarende til en arrayindeksering `arr[i]` kan man altså først tjekke om `0 <= i < n` og skrive en fejlmelding hvis dette ikke er tilfældet.

Bemærk at det kun virker hvis `arr` er erklæret og allokeret som et array, enten globalt eller lokalt, i micro-C. Det duer ikke hvis `arr` er et parameteroverført array eller hvis man laver pointerbaseret indeksering `p[i]`. Derfor er der tale om et fattigmands-indekstjek. I opgavebesvarelsen nedenfor skal du ignorere disse begrænsninger og bare håndtere det beskrevne tilfælde hvor det virker.

For at implementere denne type indekstjek skal du udvide den abstrakte maskine med en ny ordre `INDEX` der beregner adressen på et arrayelement, og du skal modificere micro-C oversætteren lidt.

Virkningen af den nye ordre kan beskrives som i tabellen side 140 i bogen *Programming Language Concepts*:

	Instruction	Stack before		Stack after	Effect
0	`CSTI` i	s	$\Rightarrow$	s, i	Push constant i
...					
26	`INDEX`	s, a, i	$\Rightarrow$	$s, s[a] + i$	Checked array indexing

I tabellen ovenfor er den eksisterende ordre 0 `CSTI` medtaget til sammenligning.

I den nye ordre 26 `INDEX` er a adressen på stakpladsen lige efter arrayets elementer, og denne plads indeholder tallet $q = s[a]$ som er adressen på arrayets element 0, sådan at adressen på arrayets element i er $s[a] + i$. Arrayets længde er således $a - q$. Ordren `INDEX` skal først tjekke at i er et lovligt indeks i arrayet. Hvis indekset er lovligt, så lægges adressen $q + i$ på stakken; og hvis indekset er ulovligt, så udskrives en fejlmeddelelse (fx med `System.out.println`) og den abstrakte maskine standses.

For at tilføje den nye ordre til den abstrakte maskine skal du tilpasse både filen `Machine.fs` og filen `Machine.java`. Vink: Se efter hvordan en eksisterende ordre, såsom `STOP`, håndteres i disse to filer.

Du skal også ændre micro-C oversætteren i `Comp.fs` sådan at koden der genereres for `arr[idx]` ikke længere er:

```
<arr> LDI <idx> ADD
```

men i stedet bliver:

```
<arr> <idx> INDEX
```

1. Vis (i udklip) de modifikationer du har lavet til `Machine.fs` og `Machine.java` og `Comp.fs` og giv en skriftlig forklaring af modifikationerne på 20–40 linjer.

2. Forklar hvordan dette bevirker at der kommer indekstjek både på arrayopslag `x = arr[i] + 2` og på array-tildelinger `arr[i] = 42`.

3. Skriv et micro-C testprogram baseret på denne skitse:

   ```
   void main(int i, int j) {
     int arr[3];
     ... initialisering af arr ...
     print arr[i];
     arr[j] = 42;
     ...
   }
   ```

   for at demonstrere at det fungerer som ventet for lovlige og ulovlige værdier af `i` og `j`.

4. Vis det komplette testprogram og resultaterne fra relevante kørsler af det. Forklar det forventede og det faktiske resultat af hver kørsel.

Opgave 4 (25 %): Typetjek af lvalue-udtryk

Skriv en funktion `tAccess` til at typetjekke lvalue-udtryk i micro-C, altså udtryk af en formerne `x` eller `*p` eller `a[i]`, svarende til abstrakt syntaks type `access` fra filen `Absyn.fs`. Typetjekfunktionen skal have denne type:

```
tAccess (access : access) (varEnv : varEnv) (funEnv : funEnv) : typ
```

hvor typerne `access` og `typ` er fra `Absyn.fs` og `varEnv` og `funEnv` er fra `Comp.fs`. Funktion `tAccess` skal returnere typen for det givne udtryk hvis det er veltypet, og ellers kaste en exception ved hjælp af funktionen `failwith "type error"` eller lignende.

Du kan antage at der allerede findes en tilsvarende funktion `tExpr` til at typetjekke rvalue-udtryk:

```
tExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : typ
```

Husk på at i C og micro-C kan man indeksere både ud fra arrays `arr[i]` og ud fra pointere `p[i]` hvor `p` har type `int*` eller en anden pointertype.

Hvis du vil typetjekke din definition af `tAccess`-funktionen med F#-oversætteren kan du fingere en tom definition af `tExpr` sådan her i filen `Comp.fs`:

```
let rec tExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : typ =
    failwith "not implemented"
and tAccess (access : access) (varEnv : varEnv) (funEnv : funEnv) : typ =
    ... din funktionsdefinition ...
```

1. Vis din `tAccess` funktion i sin helhed og skriv 15–30 linjers forklaring af den.

(Bemærk at det ikke kræves at du skal udføre `tAccess`-funktionen; det kan jo ikke gøres når der ikke er defineret en fungerende `tExpr` funktion).