Bachelor PRD **IT University, E2011**

Skriftlig eksamen, Programmer som Data
Mandag 9. januar 2012

Dette eksamenssæt har 6 sider. Tjek med det samme at du har alle siderne.

Eksamens varighed er 4 timer.

Der er fire opgaver. For at få fuldt point skal du besvare alle delopgaverne tilfredsstillende. Du må bruge alle bøger, forelæsningsnoter, forelæsningsplancher, opgavesæt, opgavebesvarelser, lommeregnere og så videre under eksamen, men ingen computere (heller ikke mobiltelefoner, PDA, iPod, iPad eller lignende) som kan udføre programmer i F# eller C# eller Java eller Scala, eller som kan kommunikere med andre enheder.

Hvis en delopgave kræver at du definerer en bestemt funktion, så må du gerne **bruge den funktion i efterfølgende delopgaver**, også selv om du ikke selv har defineret den.

Hvis en delopgave 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 det resultat som delopgaven kræver.

Opgave 1 (25 %): Regulære udtryk og automater

Opgave 1.1

Hensigten med dette regulære udtryk er at genkende tal der er skrevet med cifrene 0 til 9 og med understreg (_) korrekt anvendt som tusindtalsseparator:

```
([0-9]|[0-9][0-9]|[0-9][0-9][0-9])(_[0-9][0-9][0-9])*
```

Fx genkender det regulære udtryk strengene `123` og `1_123` og `10_000_000` men forkaster `1_23` og `1_00_000` og `_123`.

For at gøre udtrykket nemmere at læse og for at reducere skrivearbejdet i opgaverne nedenfor kan man skrive `d` i stedet for `[0-9]`:

```
(d|dd|ddd)(_ddd)*
```

Lav en ikke-deterministisk automat (NFA) der svarer til ovenstående udtryk. Du skal enten bruge en systematisk konstruktion svarende til den i forelæsningen eller Mogensens bog, eller forklare hvorfor den resulterende automat er korrekt.

Opgave 1.2

Lav en deterministisk automat (DFA) der svarer til ovenstående udtryk. Du skal enten bruge en systematisk konstruktion svarende til den i forelæsningen eller Mogensens bog, eller forklare hvorfor den resulterende automat er korrekt.

Opgave 1.3

I nogle kulturer grupperes cifre ikke nødvendigvis tre ad gangen som krævet af det regulære udtryk ovenfor. I Indien kan man fx skrive 10 millioner som `1_00_00_000` i stedet for `10_000_000`.

Lav et regulært udtryk der accepterer tal med vilkårligt mange understreger i, men sådan at der er mindst et ciffer på hver side af enhver understreg. Dvs. der må ikke komme to understreger lige efter hinanden, og tallet må ikke starte eller slutte med en understreg.

Opgave 2 (20 %): Typetjek og typeinferens

Afsnit 4.8 i *Programming Language Concepts for Software Developers* indeholder et typesystem for et simpelt funktionelt sprog.

Antag nu at dette sprog udvides med træer af heltal, hvor et træ enten er tomt og har formen `Empty`, eller har en rodknude der indeholder tallet i og venstre og højre deltræ v_1 og v_2 og da har formen $\texttt{Node}(i, v_1, v_2)$ hvor i er et heltal og v_1 og v_2 er træer.

Et udtryk, hvis værdi er et sådant træ af heltal, har type `inttree`.

Typereglerne for træ-konstruktorerne `Empty` og `Node` kunne se sådan ud:

$$\frac{}{\rho \vdash \texttt{Empty} : \texttt{inttree}}\,(empty)$$

$$\frac{\rho \vdash e_0 : \texttt{int} \qquad \rho \vdash e_1 : \texttt{inttree} \qquad \rho \vdash e_2 : \texttt{inttree}}{\rho \vdash \texttt{Node}(e_0, e_1, e_2) : \texttt{inttree}}\,(node)$$

Opgave 2.1

Lav et typeinferenstræ for udtrykket $\texttt{Node}(\texttt{11} + \texttt{10}, \texttt{Empty}, \texttt{Node}(42, \texttt{Empty}, \texttt{Empty}))$.

Opgave 2.2

Antag nu at sproget også har udtryk af formen $\texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2$ hvis værdi er værdien af e_1 dersom e evaluerer til `Empty`, og ellers er værdien af e_2 dersom e evaluerer til $\texttt{Node}(i, x_1, x_2)$. I udtrykket e_2 er variablene i og x_1 og x_2 bundet til træets bestanddele. Med andre ord virker dette `match`-udtryk ligesom pattern matching i F#.

Nedenfor er fem forslag til typeregler for `match`-udtryk. Kun en af dem er korrekt. Angiv nummeret på den korrekte regel, og forklar hvorfor det er den rigtige regel (eller hvorfor de andre regler er forkerte).

$$\frac{\rho \vdash e : \texttt{inttree} \qquad \rho \vdash e_1 : \texttt{inttree} \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto \texttt{inttree}, x_2 \mapsto \texttt{inttree}] \vdash e_2 : t}{\rho \vdash \texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2 : t}\,(1)$$

$$\frac{\rho \vdash e : \texttt{inttree} \qquad \rho \vdash e_1 : t \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto \texttt{inttree}, x_2 \mapsto \texttt{inttree}] \vdash e_2 : t}{\rho \vdash \texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2 : t}\,(2)$$

$$\frac{\rho \vdash e : \texttt{inttree} \qquad \rho \vdash e_1 : t \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto \texttt{int}, x_2 \mapsto \texttt{inttree}] \vdash e_2 : t}{\rho \vdash \texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2 : t}\,(3)$$

$$\frac{\rho \vdash e : \texttt{inttree} \qquad \rho \vdash e_1 : \texttt{inttree} \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto t, x_2 \mapsto t] \vdash e_2 : \texttt{inttree}}{\rho \vdash \texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2 : \texttt{inttree}}\,(4)$$

$$\frac{\rho \vdash e : \texttt{int} \qquad \rho \vdash e_1 : t \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto \texttt{inttree}, x_2 \mapsto \texttt{inttree}] \vdash e_2 : t}{\rho \vdash \texttt{match}\ e\ \texttt{with Empty -> } e_1\ \texttt{|}\ \texttt{Node}(i, x_1, x_2)\ \texttt{->}\ e_2 : t}\,(5)$$

Opgave 2.3

Antag at sproget også har en ny version af `let`-binding, der ligesom i F# kan splitte et træ i dets komponenter i og x_1 og x_2, sådan at i udtrykket e_b er variablene i og x_1 og x_2 bundet til træets bestanddele:

$$\texttt{let}\ \texttt{Node}(i, x_1, x_2) = e_r\ \texttt{in}\ e_b\ \texttt{end}$$

Skriv en typeregel for sådanne `let`-bindinger. (Ligesom i F# vil evalueringen af dette udtryk kaste en exception hvis e_r evaluerer til `Empty` og ikke til $\texttt{Node}(\ldots)$, men dette behøver typereglen *ikke* beskrive).

Opgave 3 (25 %): Parsing af pensionsprodukter

Denne opgave handler om et lille sprog til at beskrive pensionsprodukter. Et pensionsprodukt modelleres med et antal tilstande, såsom Active (man arbejder og indbetaler løbende til sin pension), Retired (man er holdt op at arbejde og får løbende pension udbetalt), og Dead (man er død og får ikke længere noget udbetalt).

Den forsikrede kan overgå fra en tilstand til en anden, fx fra Active til Retired; disse tilstandsovergange har navne, fx "Retirement".

Der kan være knyttet løbende betalinger til en tilstand; fx sålænge man er Active indbetaler man 50.000 kroner til sin pension per år, og så længe man er Retired får man udbetalt 20.000 kroner per måned.

Desuden kan der være knyttet en "klumpbetaling" (engangsbetaling) til overgangen fra en tilstand til en anden; fx kan man få en alderssum udbetalt ved overgang fra Active til Retired, eller få en livsforsikring udbetalt ved overgang fra Active til Dead.

Beskrivelsen af et pensionsprodukt har tre dele, som beskriver tilstandene, tilstandsovergangene, og betalingerne. Eksemplet i figur 1 viser nogle af mulighederne.

```
STATES
  Active, Retired, Dead

TRANSITIONS
  Retirement: Active --> Retired
  DieWhileActive: Active --> Dead
  DieWhileRetired: Retired --> Dead

PAYMENTS
  WHILE Active PAY 55_000 PER YEAR
  WHILE Retired RECEIVE 20_000 PER MONTH
  UPON Retirement RECEIVE 400_000
  UPON DieWhileActive RECEIVE 2_000_000
```

Figure 1: Eksempel på et pensionsprodukt.

Den abstrakte syntaks for pensionsprodukter i F# er vist nedenfor.

```
module Absyn
type direction =
    | Pay
    | Receive
type timeunit =
    | PerMonth
    | PerYear
type payment =
    | Stream of string * direction * int * timeunit
    | Lumpsum of string * direction * int
type state = string
type transition = string * (state * state)
type pension = state list * transition list * payment list
```

Et eksempel på denne abstrakte syntaks ses i opgave 3.2.

Opgave 3.1

Skriv regeldelen af en parserspecifikation til `fsyacc` for pensionsprodukter, sådan at den i hvert fald kan parse eksemplet i figur 1. Du behøver ikke skrive de semantiske aktioner i dette spørgsmål.

Du kan antage at der er defineret tokens (fx `STATES`) svarende til de forskellige nøgleord og svarende til de forskellige øvrige tegn. Token `NAME` svarer til et navn (på en tilstand eller overgang) og token `AMOUNT` svarer til et tal såsom `50_000`:

```
%token <string> NAME
%token <int> AMOUNT
%token MONTH PAY PAYMENTS PER RECEIVE STATES TRANSITIONS UPON WHILE YEAR
%token ARROW COLON COMMA
%token EOF
```

Opgave 3.2

Udvid parserspecifikationen fra delopgave 3.1 med semantiske aktioner indeholdt i `{ ... }`, sådan at parseren konstruerer abstrakt syntaks af type `pension` svarende til den konkrete syntaks.

For pensionsproduktet i figur 1 skal der for eksempel produceres denne abstrakte syntaks, af type `pension`:

```
(["Active"; "Retired"; "Dead"],
 [("Retirement", ("Active", "Retired"));
  ("DieWhileActive", ("Active", "Dead"));
  ("DieWhileRetired", ("Retired", "Dead"))],
 [Stream ("Active",Pay,55000,PerYear);
  Stream ("Retired",Receive,20000,PerMonth);
  Lumpsum ("Retirement",Receive,400000);
  Lumpsum ("DieWhileActive",Receive,2000000)])
```

Opgave 3.3

Angiv et fragment af lexerspecifikationen til `fslex` som kan scanne et beløb (svarende til token `AMOUNT`) fra et tal angivet med korrekt anbragte tusindseparatorer. I denne delopgave skal du kun skrive venstresiden af en lexerregel, dvs. den del der matcher et tal med tusindseparator.

Opgave 3.4

Skriv højresiden af lexerreglen svarende til opgave 3.3, dvs. et F#-udtryk der fremstiller et `AMOUNT` token indeholdende det heltal som fremkommer ved at konvertere den scannede streng (fx `50_000`). Definér gerne en hjælpefunktion til dette formål. Vink: I F# kan man bruge `s.[i]` til at udtrække `i`'te tegn fra strengen `s`, hvor tegnene er nummereret fra 0.

Opgave 4 (30 %): F#-funktioner til at forbedre bytekode

I kursets noter PLCSD defineres en abstrakt maskine hvis instruktioner (bytekode) kan beskrives som en datatype i F#, vist i figur 2.

```
type label = string
type instr =
  | Label of label                    (* symbolic label; pseudo-instruc. *)
  | CSTI of int                       (* constant                        *)
  | ADD                               (* addition                        *)
  | SUB                               (* subtraction                     *)
  | MUL                               (* multiplication                  *)
  | DIV                               (* division                        *)
  | MOD                               (* modulus                         *)
  | EQ                                (* equality: s[sp-1] == s[sp]      *)
  | LT                                (* less than: s[sp-1] < s[sp]      *)
  | NOT                               (* logical negation:  s[sp] != 0   *)
  | DUP                               (* duplicate stack top             *)
  | SWAP                              (* swap s[sp-1] and s[sp]          *)
  | LDI                               (* get s[s[sp]]                    *)
  | STI                               (* set s[s[sp-1]]                  *)
  | GETBP                             (* get bp                          *)
  | GETSP                             (* get sp                          *)
  | INCSP of int                      (* increase stack top by m         *)
  | GOTO of label                     (* go to label                     *)
  | IFZERO of label                   (* go to label if s[sp] == 0       *)
  | IFNZRO of label                   (* go to label if s[sp] != 0       *)
  | CALL of int * label               (* move m args up 1, push pc, jump *)
  | TCALL of int * int * label        (* move m args down n, jump        *)
  | RET of int                        (* pop m and return to s[sp]       *)
  | PRINTI                            (* print s[sp] as integer          *)
  | PRINTC                            (* print s[sp] as character        *)
  | LDARGS                            (* load command line args on stack *)
  | STOP                              (* halt the abstract machine       *)
```

Figure 2: Bytekodeinstruktioner for stakmaskine fra PLCSD.

De følgende opgaver går ud på at skrive F#-funktioner der kan forbedre bytekodesekvenser, for eksempel så de bruger færre instruktioner til at udrette det samme. For eksempel kunne `[LDI; CSTI 0; ADD; ...]` forenkles til `[LDI; ...]` idet det ikke har nogen effekt at lægge 0 til et tal.

Opgave 4.1

Figur 3 viser en række mulige forenklinger af bytekode, gengivet fra PLCSD afsnit 12.3. Find på yderligere tre forbedringer af lignende art. Overvej fx hvad `[CSTI 17; CSTI 42; ADD]` kunne forbedres til.

Opgave 4.2

Definér en F# funktion `simplify : instr list -> instr list` der forbedrer bytekodesekvenser så de bruger færre instruktioner til at udrette det samme. For eksempel bør `simplify [LDI; CSTI 0; ADD; ...]` give `[LDI; ...]`.

Det er nok at din `simplify`-funktion implementerer 6 af de forbedringer der er vist i figur 3.

Opgave 4.3

Definér en F# funktion `simplifyAll : instr list -> instr list` der kalder `simplify` på en bytekodesekvens indtil den ikke kan foretage flere forbedringer.

Opgave 4.4

Definér en F# funktion `removeDead : instr list -> instr list` der fjerner døde instruktioner. Det vil sige at den fjerner alle de bytekodeinstruktioner der ligger mellem et ubetinget hop (`GOTO` eller `RET`) og den

`CSTI 0, EQ`	has the same meaning as	`NOT`	
`CSTI 0, ADD`	has the same meaning as	⟨empty⟩	
`CSTI 0, SUB`	has the same meaning as	⟨empty⟩	
`CSTI 0, NOT`	has the same meaning as	`CSTI 1`	
`CSTI` n`, NOT`	has the same meaning as	`CSTI 0`	when $n \neq 0$
`CSTI 1, MUL`	has the same meaning as	⟨empty⟩	
`CSTI 1, DIV`	has the same meaning as	⟨empty⟩	
`CSTI` n`, INCSP` m	has the same meaning as	`INCSP` $(m+1)$	when $m < 0$
`CSTI 0, IFZERO` a	has the same meaning as	`GOTO` a	
`CSTI` n`, IFZERO` a	has the same meaning as	⟨empty⟩	when $n \neq 0$
`CSTI 0, IFNZRO` a	has the same meaning as	⟨empty⟩	
`CSTI` n`, IFNZRO` a	has the same meaning as	`GOTO` a	when $n \neq 0$
`NOT, NOT`	has the same meaning as	⟨empty⟩	
`NOT, IFZERO` a	has the same meaning as	`IFNZRO` a	
`NOT, IFNZRO` a	has the same meaning as	`IFZERO` a	
`INCSP 0`	has the same meaning as	⟨empty⟩	
`INCSP` m_1`, INCSP` m_2	has the same meaning as	`INCSP` $(m_1 + m_2)$	
`INCSP` m_1`, RET` m_2	has the same meaning as	`RET` $(m_2 - m_1)$	

Figure 3: Nogle forbedringer af bytekodesekvenser, fra PLCSD afsnit 12.3

første label efter det ubetingede hop. Sådanne instruktioner vil nemlig aldrig blive udført.

For eksempel skal `removeDead [CSTI 42; GOTO "L1"; CSTI 17; ADD; Label "L2"; PRINTI]` give `[CSTI 42; GOTO "L1"; Label "L2"; PRINTI]`.

Opgave 4.5

Definér en funktion `usedLabels : instr list -> string list` der returnerer en liste af alle de labels der bliver brugt af `GOTO`, `IFZERO` eller `IFNZRO` instruktioner i den givne bytekodesekvens. Det er uden betydning om den resulterende liste har dubletter eller ej.

Opgave 4.6

Definér en funktion `removeUnusedLabels : instr list -> instr list` der fjerner labels som ikke bruges af nogen `GOTO`, `IFZERO` eller `IFNZRO` instruktion.