BSWU BPRD **IT University, E2012**

Skriftlig eksamen, Programmer som Data
Torsdag 3. januar 2013

Version 1.1 af 2013-01-03

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) der 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.

Baggrund for opgavesættet

Industrielt udstyr og maskiner kan styres af software ved at man tilslutter analoge og digitale indgange og udgangen til computeren.

Et udbredt enhed til styring af apparater er K8055-kortet, et hobbyprodukt fra firmaet Velleman. K8055-kortets ind- og udgange forbindes til et apparat eller andre dimser, og kortet selv forbindes med en almindelig computer via et USB-kabel; se eksemplet i figur 1.

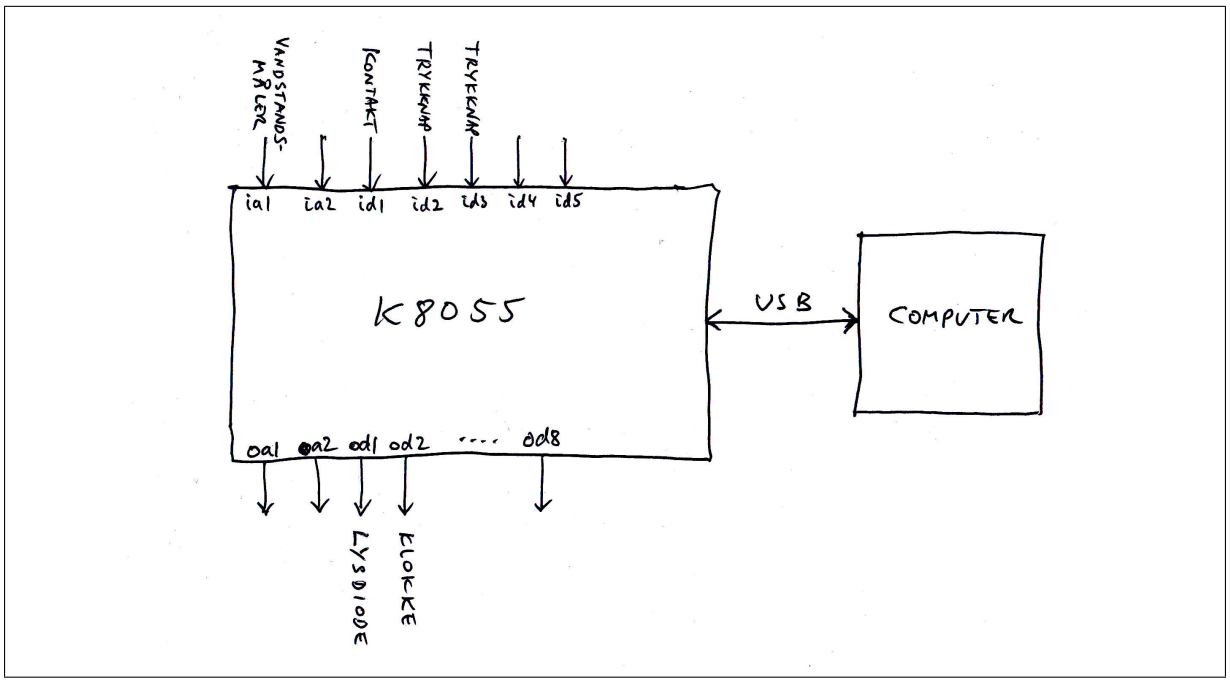

Figure 1: Kontakter, lysdioder og andre dimser; K8055-kort; og computer.

K8055-kortet har følgende *indgange* (som kan aflæse kontakter og måleinstrumenter) og *udgange* (som kan aktivere lamper, motorer, osv.):

- To analoge indgange `ia1` og `ia2`. Deres mulige værdier er heltallene fra 0 til 255.
- Fem digitale indgange `id1`, `id2`, `id3`, `id4` og `id5`. Deres mulige værdier er TRUE og FALSE.
- To tællere `cd1` og `cd2`, der tæller hvor mange gange `id1` og `id2` har skiftet fra FALSE til TRUE. Deres mulige værdier er heltallene fra 0 til 65535.
- To analoge udgange `oa1` og `oa2`. Deres mulige værdier er heltallene fra 0 til 255.
- Otte digitale udgange `od1`, `od2`, `od3`, `od4`, `od5`, `od6`, `od7` og `od8`. Deres mulige værdier er TRUE og FALSE.

Alle indgangene kan opfattes som variable hvis værdier styres af omgivelserne (fx termometre eller kontakter der er monteret på et apparat der skal styres); disse værdier kan aflæses af software på computeren. Alle udgangene kan opfattes som variable hvis værdier tildeles af software på computeren; disse værdier påvirker så det apparat der skal styres, fx ved at en digital udgang tænder og slukker for en lysdiode, eller ved at en analog udgang styrer hvor hurtigt en motor kører.

Resten af denne opgave handler om et meget simpelt sprog kaldet SIMPLC til at styre apparater ved brug af de ovennævnte analoge og digitale ind- og udgange. Sprogets navn kommer af at apparatstyringer ofte kaldes "programmable logic controllers" (PLC).

Formen af SIMPLC-programmer

Et *program* i SIMPLC består af en sekvens af blokke, hvor en *blok* er en *label* efterfulgt af en sekvens af *kommandoer*. Der er fire slags kommandoer `cmd`:

- Tildeling (assignment) til en analog eller digital udgang, af formen `output := expr`. Det udføres ved at udregne værdien af udtrykket `expr` og skrive den til udgangen `output`, fx `oa1`.
- Ubetinget hop, af formen `GOTO lab`.
- Betinget hop, af formen `IF expr GOTO lab`.
- En venteordre, af formen `SLEEP n`, hvor konstanten `n` angiver et antal millisekunder.

Et *udtryk* `expr` i SIMPLC har en af disse former:

- En logisk konstant: `FALSE` eller `TRUE`.
- En ikke-negativ heltalskonstant: `0`, `1`, ...
- Aflæsning af en analog indgang (fx `ia1`) eller en digital indgang (fx `id1`) eller en tæller (fx `cd1`).
- Et aritmetisk udtryk `e1+e2` eller `e1-e2`.
- Et sammenligningsudtryk `e1=e2` eller `e1<>e2` eller `e1<e2`.
- Et sammensat logisk udtryk: `NOT e1` eller `e1 AND e2` eller `e1 OR e2`.

Eksempler på SIMPLC-programmer

Dette SIMPLC-program sætter digital udgang nummer 1 til sand, venter 500 millisekunder, sætter den til falsk, venter, og begynder forfra. Hvis digital udgang 1 er tilsluttet en lysdiode, så vil man se den blinke med en periode på 1 sekund:

```
L1: od1 := TRUE;
    SLEEP 500;
    od1 := FALSE;
    SLEEP 500;
    GOTO L1;
```

Dette SIMPLC-program sætter digital udgang nummer 1 til værdien af digital indgang 1, venter 500 ms, sætter den til falsk, venter, og begynder forfra. Hvis digital indgang 1 er tilsluttet en kontakt og digital udgang 1 er tilsluttet en lysdiode, vil man se lysdioden blinke så længe kontakten er sluttet:

```
L1: od1 := id1;
    SLEEP 500;
    od1 := FALSE;
    SLEEP 500;
    GOTO L1;
```

Dette SIMPLC-program sætter digital udgang nummer 2 til sand hvis digital indgang 1 er sand og værdien på analog indgang 1 ikke er mindre end 128 (dvs. er større end eller lig 128), venter 10 millisekunder, og begynder så forfra. Hvis `id1` er tilsluttet en kontakt, `ia1` en vandstandsmåler, og `od2` en klokke, så ringer klokken så længe kontakten er sluttet og vandstanden er for høj:

```
L1: od2 := id1 AND NOT (ia1 < 128);
    SLEEP 10;
    GOTO L1;
```

Dette SIMPLC-program fungerer omtrent som ovennævnte, men et tryk på trykknappen sluttet til `id2` vil nu slå vandstandsovervågningen fra, og et tryk på trykknappen sluttet til `id3` vil slå den til igen:

```
On:  od2 := NOT (ia1 < 128);
     SLEEP 10;
     IF id2 GOTO Off;
     GOTO On;
Off: od2 := FALSE;
     SLEEP 10;
     IF id3 GOTO On;
     GOTO Off;
```

Opgave 1 (25 %): Lexerspecifikation for SIMPLC

Opgave 1.1

Skriv selv et SIMPLC program der ringer med klokken (på digital udgang 2) når der 14 gange har været trykket på trykknappen ved digital indgang 2. Vink: Brug tæller 2. Der skal ventes 100 millisekunder mellem hvert tjek af tælleren.

Opgave 1.2

Skriv en lexer-specifikation for SIMPLC, der kan bruges som input til `fslex`. Det er nok at skrive regel-delen, dvs. den del der starter med `rule Token = ...`.

Eventuelle nødvendige hjælpefunktioner kan du nøjes med at beskrive uden at programmere dem i F#.

Det er hensigtsmæssigt at lexeren genkender navne på indgange (`ia1`, `id1`, `cd1` osv) og udgange (`oa1`, `od1`, osv) i lexeren, og genererer tokens `INANA`, `INDIGI`, `INCOUNT`, `OUTANA` og `OUTDIGI`, hvor hver token indeholder ind/udgangens nummer (som kan have værdierne 1–2, 1–5, 1–2, 1–2, og 1–8 i de fem tokens).

Du kan antage at parserspecifikationen definerer følgende tokens:

```
%token <string> NAME
%token <int> CSTINT INANA INDIGI INCOUNT OUTANA OUTDIGI
%token COLON SEMICOLON COLONEQUAL
%token FALSE TRUE
%token IF GOTO SLEEP
%token PLUS MINUS
%token EQ NE LT
%token AND NOT OR
%token LPAR RPAR
%token EOF
```

Opgave 2 (25 %): Parserspecifikation for SIMPLC

Denne opgave handler om en parserspecifikation til SIMPLC, som kunne bruges som input til `fsyacc`.

Opgave 2.1

Skriv præcedenserklæringerne til en parserspecifikation for SIMPLC, sådan at aritmetik, sammenligninger og logiske operatorer får samme præcedens som kendes fra Java eller C#.

Opgave 2.2

Skriv en parserspecifikation for alle dele af grammatikken for SIMPLC, dvs. programmer, blokke, kommandoer og udtryk, men udelad de semantiske aktioner.

Opgave 2.3

Skriv nu også semantiske aktioner til parserspecifikationen fra foregående opgave, så der konstrueres abstrakt syntaks (i F#) for SIMPLC-programmer ved brug af disse typer:

```
type expr =
  | Ia of int         // 1-2
  | Id of int         // 1-5
  | Cd of int         // 1-2
  | False
  | True
  | CstI of int
  | Prim1 of string * expr
  | Prim2 of string * expr * expr

type output =
  | Oa of int         // 1-2
  | Od of int         // 1-8

type lab = string

type cmd =
  | Set of output * expr
  | Goto of lab
  | If of expr * lab  // if true go to lab
  | Sleep of int      // milliseconds

type program = (lab * cmd list) list
```

For eksempel vil den abstrakte syntaks (af type `program`) for det første SIMPLC-eksempel på side 2 se sådan ud:

```
[("L1", [Set (Od 1,True); Sleep 500; Set (Od 1,False); Sleep 500; Goto "L1"])]
```

Opgave 3 (30 %): Typeregler og andre tjek i SIMPLC

For at undgå fejltagelser såsom `od1 := ia1`, der aflæser en analog indgang (0–255) og forsøger at skrive dens værdi til en digital udgang (`TRUE` eller `FALSE`), forsyner vi nu SIMPLC med et typesystem.

I SIMPLC bruges typerne `BOOL`, `BYTE` og `WORD`.

Analoge indgange har type `BYTE`, digitale indgange har type `BOOL`, og tællerne `cd1` og `cd2` samt heltalskonstanter har type `WORD`. Til analoge udgange [her stod fejlagtigt: "indgange"] kan man skrive værdier af type `BYTE` og `WORD` (værdien modulo 256 skrives), og til digitale udgange kan man skrive værdier af type `BOOL`.

Aritmetiske udtryk forventer værdier af type `BYTE` eller `WORD` og producerer værdier af type `WORD`. Sammenligninger forventer værdier af type `BYTE` eller `WORD` og producerer værdier af type `BOOL`. Logiske operatorer forventer værdier af type `BOOL` og producerer værdier af type `BOOL`.

Opgave 3.1

Lav typeregler for udtryk i SIMPLC-sproget, dvs. for operatorerne `AND`, `OR`, `NOT`, `PLUS`, `MINUS`, `EQ`, `NE`, `LT`, indgangene `ia`j, `id`j, `cd`j, og konstanterne `FALSE`, `TRUE` og ikke-negative heltal.

Bemærk at eftersom sproget ikke har variable og funktioner, er der ikke brug for omgivelserne (environment) ρ, så et "type judgement" har formen $\vdash e : t$ hvor e er et udtryk og t en type.

For eksempel vil typereglen for `NOT e` se sådan ud:

$$\frac{\vdash e : \texttt{BOOL}}{\vdash \texttt{NOT}\ e : \texttt{BOOL}}$$

Opgave 3.2

Skriv en F#-funktion `exprType : expr -> typ` der tjekker at et SIMPLC-udtryk er veltypet, og i så fald returnerer dets type, og ellers kaster en exception.

F#-typen `typ` er erklæret således:

```
type typ = BOOL | BYTE | WORD
```

Opgave 3.3

Skriv en F#-funktion `cmdType : cmd -> bool` der tjekker at en SIMPLC-kommando er veltypet, og i så fald returnerer `true`, og ellers kaster en exception.

Dette går væsentligst ud på at tjekke at tildelinger til analoge udgange har type `BYTE` eller `WORD`, at tildelinger til digitale udgange har type `BOOL`, og at betingelsen i en `IF`-kommando har type `BOOL`.

Opgave 3.4

Skriv en F#-funktion `labelCheck : program -> bool` der tjekker at alle de labels der bruges i `GOTO`- og `IF-GOTO`-kommandoer faktisk er defineret i programmet.

Opgave 4 (20 %): Scala

I denne opgave skal du bruge programmeringssproget Scala i stedet for F#.

Opgave 4.1

Brug Scalas "case classes" til at erklære abstrakt syntaks for SIMPLC svarende til alle F#-erklæringerne for den abstrakte syntaks vist i opgave 2.3.

For eksempel skal du erklære en abstrakt Scala-klasse `Expr` svarende til F#-type `expr`, en abstrakt Scala-klasse `Output` svarende til F#-typen `output`, og så videre.

Vink: Ligesom F# tillader Scala typeforkortelser såsom `type Lab = String`.

Opgave 4.2

Skriv en Scala-funktion `simplify(e: Expr): Expr` der forenkler *logiske* deludtryk, repræsenteret i den abstrakte syntaks fra opgave 4.1. Du kan antage at udtrykket kun består af digitale indgange (`id1` osv), konstanter, og logiske operatorer, og ingen regneoperatorer eller sammenligningsoperatorer.

Din funktion skal lave så mange forskellige forenklinger som muligt, men mindst syv. Figur 2 viser nogle ideer til forenkling af logiske udtryk.

`FALSE AND e2`	forenkles til	`FALSE`
`TRUE AND e2`	forenkles til	`e2`
`NOT TRUE`	forenkles til	`FALSE`
`NOT(NOT e)`	forenkles til	`e`

Figure 2: Nogle forenklinger af logiske udtryk i SIMPLC.