# Løsningsforslag
# Skriftlig eksamen
# 3. januar 2013

Version 1, 2013-01-03

## Spørgsmål 1

### Spørgsmål 1.1

```
L1: od2 := FALSE;
L2: SLEEP 100;
    IF (cd2 < 14) GOTO L2;
    od2 := TRUE;
```

Ovenstående løser opgaven fordi digital udgang 2 (klokken) sættes til false (slukket) og forbliver uændret indtil `cd2`, som tæller antal gange digital indgang 2 er skiftet fra false til true, får en værdi der er større end eller lig med 14, hvor den så sættes til true, og SIMPLC-programmet standser (men digital udgang 2 forbliver true og klokken bliver ved med at ringe).

En anden mulig løsning er denne, som sparer en label, men lidt overflødigt sætter digital udgang 2 til false igen og igen:

```
L1: od2 := FALSE;
    SLEEP 100;
    IF (cd2 < 14) GOTO L1;
    od2 := TRUE;
```

### Spørgsmål 1.2

Her er en komplet regel til lexerspecifikationen, der bruger regulære udtryk til kun at genkende de tilladte navne på ind- og udgange:

```
rule Token = parse
  | [' ' '\t' '\r'] { Token lexbuf }
  | '\n'            { lexbuf.EndPos <- lexbuf.EndPos.NextLine; Token lexbuf }
  | ['0'-'9']+      { CSTINT (System.Int32.Parse (lexemeAsString lexbuf)) }
  | "ia"['1'-'2']   { INANA   (lexemeAsString lexbuf |> parseThirdInt) }
  | "id"['1'-'5']   { INDIGI  (lexemeAsString lexbuf |> parseThirdInt) }
  | "cd"['1'-'2']   { INCOUNT (lexemeAsString lexbuf |> parseThirdInt) }
  | "oa"['1'-'2']   { OUTANA  (lexemeAsString lexbuf |> parseThirdInt) }
  | "od"['1'-'8']   { OUTDIGI (lexemeAsString lexbuf |> parseThirdInt) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']* { keyword (lexemeAsString lexbuf) }
  | ':'             { COLON      }
  | ';'             { SEMICOLON  }
  | ":="            { COLONEQUAL }
  | '+'             { PLUS  }
  | '-'             { MINUS }
  | '='             { EQ    }
  | '<'             { LT    }
  | '('             { LPAR  }
  | ')'             { RPAR  }
  | "<>"            { NE    }
  | eof             { EOF   }
  | _               { failwith "Lexer error: illegal symbol" }
```

Ovenstående har den konsekvens at `od8` ikke vil være et tilladt labelnavn (fordi det er navnet på en digital udgang), mens `od9` er et tilladt labelnavn (fordi det ikke er navnet på en digital udgang). Om dette er et godt design er en smagssag, men det svarer til at `while` ikke kan bruges som label i C# hvorimod `during` kan.

Her er de hjælpefunktioner som benyttes ovenfor:

```
let keyword s =
    match s with
    | "AND"         -> AND
    | "FALSE"       -> FALSE
    | "GOTO"        -> GOTO
    | "IF"          -> IF
    | "NOT"         -> NOT
    | "OR"          -> OR
    | "TRUE"        -> TRUE
    | "SLEEP"       -> SLEEP
    | _             -> NAME s

let parseThirdInt (s : string) =
    System.Int32.Parse(s.Substring(2))
```

# Spørgsmål 2

## Spørgsmål 2.1

Traditionelt (og i C, C++, Java, C#) binder "eller" svagere end "og", som binder svagere end "not", som binder svagere end "=" og "<>", som binder svagere end "<", som binder svagere end "+" og "-":

```
%left OR              /* lowest precedence  */
%left AND
%nonassoc NOT
%left EQ NE
%nonassoc LT
%left PLUS MINUS      /* highest precedence  */
```

## Spørgsmål 2.2 og 2.3

Her er parserspecifikationen inklusive semantiske aktioner, i to afdelinger. For det første består et SIMPLC-program af en sekvens (`Blocks`) af blokke; hver block (`Block`) er en label efterfulgt af en sekvens (`Commands`) af kommandoer efterfulgt af semikolon; hver kommando (`Command`) har en af de fire angivne former; og en udgang (`Output`) har en af de to former (analog, digital):

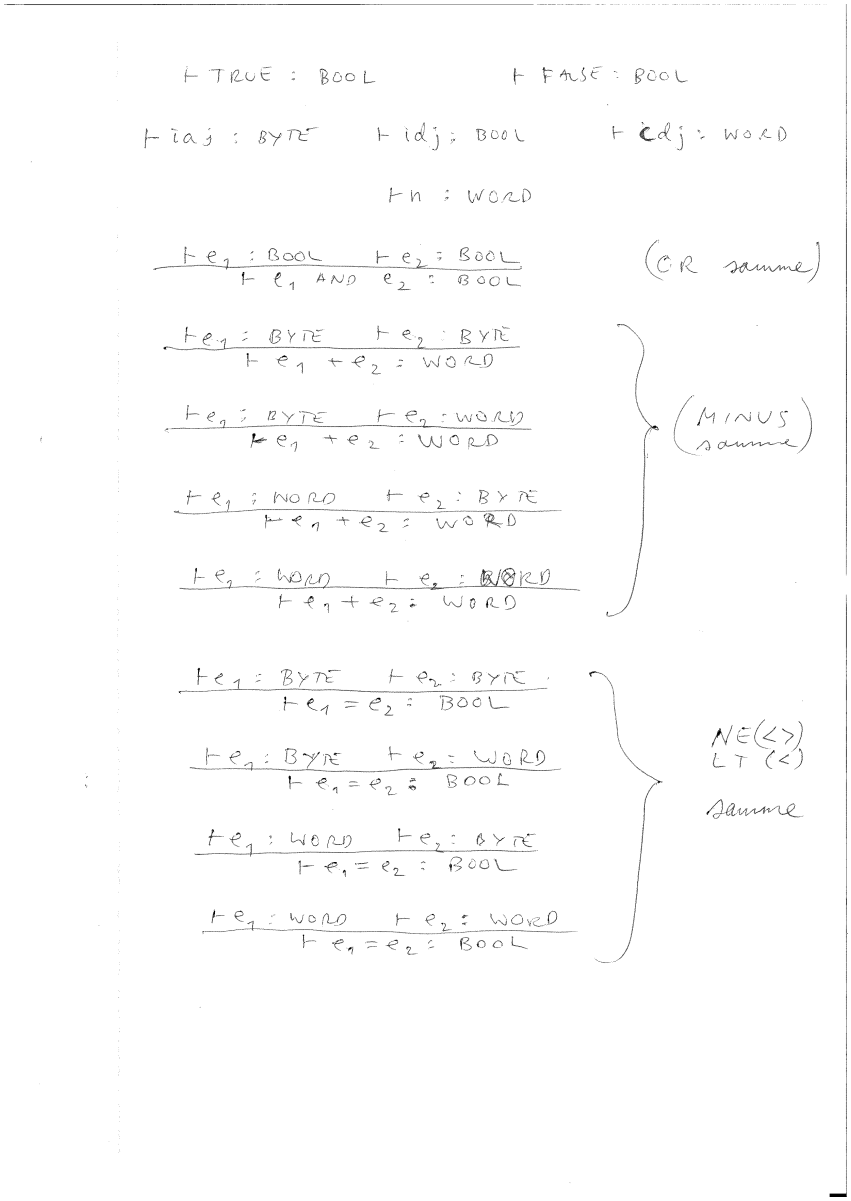
```
Main:
    Blocks EOF                          { $1 }
;
Blocks:
    /* empty */                         { [] }
  | Block Blocks                        { $1 :: $2 }
;
Block:
    NAME COLON Commands                 { ($1, $3) }
;
Commands:
    /* empty */                         { [] }
  | Command SEMICOLON Commands          { $1 :: $3 }
;
Command:
    Output COLONEQUAL Expr              { Set($1, $3) }
  | IF Expr GOTO NAME                   { If($2, $4)  }
  | GOTO NAME                           { Goto($2)    }
  | SLEEP CSTINT                        { Sleep($2)   }
;
Output:
   OUTANA                              { Oa($1) }
 | OUTDIGI                             { Od($1) }
;
```

For det andet har et udtryk (`Expr`) en af de angivne 13 former, hvor en indgang (`Input`) kan antage en af de 3 former (analog, digital, tæller):

```
Expr:
   CSTINT                          { CstI $1              }
 | FALSE                           { False                }
 | TRUE                            { True                 }
 | Input                           { $1                   }
 | Expr PLUS Expr                  { Prim2("+", $1, $3)   }
 | Expr MINUS Expr                 { Prim2("-", $1, $3)   }
 | Expr EQ Expr                    { Prim2("=", $1, $3)   }
 | Expr NE Expr                    { Prim2("<>", $1, $3)  }
 | Expr LT Expr                    { Prim2("<", $1, $3)   }
 | Expr AND Expr                   { Prim2("and", $1, $3) }
 | Expr OR Expr                    { Prim2("or", $1, $3)  }
 | NOT Expr                        { Prim1("not", $2)     }
 | LPAR Expr RPAR                  { $2                   }
;
Input:
   INANA                           { Ia($1) }
 | INDIGI                          { Id($1) }
 | INCOUNT                         { Cd($1) }
;
```

# Spørgsmål 3

## Spørgsmål 3.1




Der skal være fire regler for hver af +, −, =, <> og < for at dække alle fire kombinationer af operandtyperne `BYTE` og `WORD`.

Til gengæld er reglerne for − de samme som for +, og tilsvarende for <> og <.

Man kan faktisk nøjes med én regel per operator hvis man indfører en hjælperegel der siger at ethvert udtryk af type `BYTE` også kan opfattes som et udtryk af type `WORD`:

$$\frac{\vdash e : \texttt{BYTE}}{\vdash e : \texttt{WORD}}$$

### Spørgsmål 3.2

Her er typetjekkeren (skrevet i F#) for udtryk:

```
let rec exprType (e : expr) : typ =
    match e with
     | Ia _ -> BYTE
     | Id _ -> BOOL
     | Cd _ -> WORD
     | False -> BOOL
     | True -> BOOL
     | CstI n -> if 0 <= n && n < 255 then BYTE else WORD
     | Prim1(op, e1) ->
       match (op, exprType e1) with
       | ("not", BOOL) -> BOOL
       | ("not", _)    -> failwith "type error in not"
       | _ -> failwithf "unknown operator: %s" op
     | Prim2(op, e1, e2) ->
       match (op, exprType e1, exprType e2) with
        | ("and", BOOL, BOOL) -> BOOL
        | ("or", BOOL, BOOL)  -> BOOL
        | ("+", BYTE, BYTE)   -> WORD
        | ("+", BYTE, WORD)   -> WORD
        | ("+", WORD, BYTE)   -> WORD
        | ("+", WORD, WORD)   -> WORD
        | ("+", _, _)         -> failwith "type error in +"
        | ("-", BYTE, BYTE)   -> WORD
        | ("-", BYTE, WORD)   -> WORD
        | ("-", WORD, BYTE)   -> WORD
        | ("-", WORD, WORD)   -> WORD
        | ("-", _, _)         -> failwith "type error in -"
        | ("<", BYTE, BYTE)   -> BOOL
        | ("<", BYTE, WORD)   -> BOOL
        | ("<", WORD, BYTE)   -> BOOL
        | ("<", WORD, WORD)   -> BOOL
        | ("<", _, _)         -> failwith "type error in <"
        | ("=", BYTE, BYTE)   -> BOOL
        | ("=", BYTE, WORD)   -> BOOL
        | ("=", WORD, BYTE)   -> BOOL
        | ("=", WORD, WORD)   -> BOOL
        | ("=", _, _)         -> failwith "type error in ="
        | ("<>", BYTE, BYTE)  -> BOOL
        | ("<>", BYTE, WORD)  -> BOOL
        | ("<>", WORD, BYTE)  -> BOOL
        | ("<>", WORD, WORD)  -> BOOL
        | ("<>", _, _)        -> failwith "type error in <>"
        | _ -> failwithf "unknown operator: %s" op
```

Det er svært træls med de mange regler for plus, minus, mindre end, lig med og forskellig fra. Som i 3.1 kan man i stedet skrive fx at "reglerne for minus er lige som for plus" og "reglerne for ligmed og forskellig fra er lige som for mindre end" og dermed spare 15 linjer.

Alle gentagelserne kunne også undgås ved at bruge F#'s "or"-patterns, som i

```
match (op, exprType e1, exprType e2) with
 ...
 | ("+", (BYTE | WORD), (BYTE | WORD)) -> WORD
 | ("+", _, _) -> failwith "type error in +"
 | ("-", (BYTE | WORD), (BYTE | WORD)) -> WORD
 | ("-", _, _) -> failwith "type error in -"
 ...
```

eller endda dette, der håndterer alle plus- og minus-regler i én linje (og stadig giver en rimelig fejlmeddelelse):

```
match (op, exprType e1, exprType e2) with
 ...
 | (("+" | "-"), (BYTE | WORD), (BYTE | WORD)) -> WORD
 | ("+", _, _) -> failwith "type error in +"
 | ("-", _, _) -> failwith "type error in -"
 ...
```

og tilsvarende for sammenligningsoperatorerne, så 25 linjer reduceres til 7. Denne F#-feature var ikke en del af pensum men det er naturligvis tilladt at bruge den.

## Spørgsmål 3.3

Typetjek af en kommando er meget nemmere (men bruger selvfølgelig `exprType` som subrutine). For eksempel er en `Sleep`-kommando automatisk typekorrekt, fordi lexer og parser garanterer at argumentet er en heltalskonstant.

```
let cmdType (c : cmd) : bool =
    match c with
     | Set(out, e) ->
       match (out, exprType e) with
        | (Oa _, BYTE) -> true
        | (Oa _, WORD) -> true
        | (Od _, BOOL) -> true
        | _            -> failwith "type error in assignment"
     | Goto _   -> true
     | If(e, _) -> if exprType e = BOOL then true
                   else failwith "type error in IF"
     | Sleep _  -> true
```

## Spørgsmål 3.4

Funktionen `labelCheck` tager et argument `blocks` (som har type `program`, dvs. det er en liste af blokke), producerer en liste af alle definerede labels fra `blocks`, definerer en hjælpefunktion `labelExists(lab)` der afgør om `lab` er i listen, definerer en hjælpefunktion `checkCmd(c)` der afgør om de labels der bruges i kommando `c` er definerede, og kalder så den på alle kommandoer i alle blokke i `blocks`:

```
let labelCheck blocks =
    let labels = List.map (function (lab, _) -> lab) blocks
    let labelExists lab = List.exists (fun x -> x=lab) labels
    let checkCmd (c : cmd) =
        match c with
        | Goto lab   -> labelExists lab
        | If(_, lab) -> labelExists lab
        | _          -> true
    let checkBlock (_, cmds) = List.forall checkCmd cmds
    List.forall checkBlock blocks
```

# Spørgsmål 4

## Spørgsmål 4.1

Scala-erklæringerne for SIMPLC abstrakt syntaks er nærmest en linje-for-linje oversættelse af F#-erklæringerne vist i opgave 2.3. Men alle detaljer er anderledes; for eksempel skal felter have navne (fx `index`) i Scala mens de er navnløse i F#:

```
sealed abstract class Expr
case class Ia(index: Int) extends Expr  // 1-2
case class Id(index: Int) extends Expr  // 1-5
case class Cd(index: Int) extends Expr  // 1-2
case class CFalse() extends Expr
case class CTrue() extends Expr
case class Prim1(op: String, e1: Expr) extends Expr
case class Prim2(op: String, e1: Expr, e2: Expr) extends Expr

sealed abstract class Output
case class Oa(index: Int) extends Output   // 1-2
case class Od(index: Int) extends Output   // 1-8

type Lab = String

sealed abstract class Cmd
case class Set(lhs: Output, rhs: Expr) extends Cmd
case class Goto(lab: Lab) extends Cmd
case class If(cond: Expr, lab: Lab) extends Cmd
case class Sleep(ms: Int) extends Cmd

type Program = List[(Lab, List[Cmd])]
```

## Spørgsmål 4.2

Her er en rimelig omfattende forenklingsfunktion til logiske udtryk fra SIMPLC. Som i opgaveformuleringen tager den kun digitale indgange, konstanter og logiske operatorer (altså ikke `ia`*j*, `cd`*j*, +, −, =, <>, <) i betragtning:

```
def simplify(e: Expr) : Expr =
  e match {
    case Prim2("and", CFalse(), e2)    => CFalse()
    case Prim2("and", CTrue(), e2)     => simplify(e2)
    case Prim2("and", e1, CFalse())    => CFalse()
    case Prim2("and", e1, CTrue())     => simplify(e1)
    case Prim2("and", e1, e2)          => Prim2("and", simplify(e1), simplify(e2))
    case Prim2("or", CFalse(), e2)     => simplify(e2)
    case Prim2("or", CTrue(), e2)      => CTrue()
    case Prim2("or", e1, CFalse())     => simplify(e1)
    case Prim2("or", e2, CTrue())      => CTrue()
    case Prim2("or", e1, e2)           => Prim2("or", simplify(e1), simplify(e2))
    case Prim1("not", Prim1("not", e)) => simplify(e)
    case Prim1("not", CFalse())        => CTrue()
    case Prim1("not", CTrue())         => CFalse()
    case Prim1("not", Prim2("and", e1, e2)) =>
      simplify(Prim2("or", simplify(Prim1("not", e1)), simplify(Prim1("not", e2))))
    case Prim1("not", Prim2("or", e1, e2))  =>
      simplify(Prim2("and", simplify(Prim1("not", e1)), simplify(Prim1("not", e2))))
    case Prim1("not", e)               => Prim1("not", simplify(e))
    case _ => e
  }
```