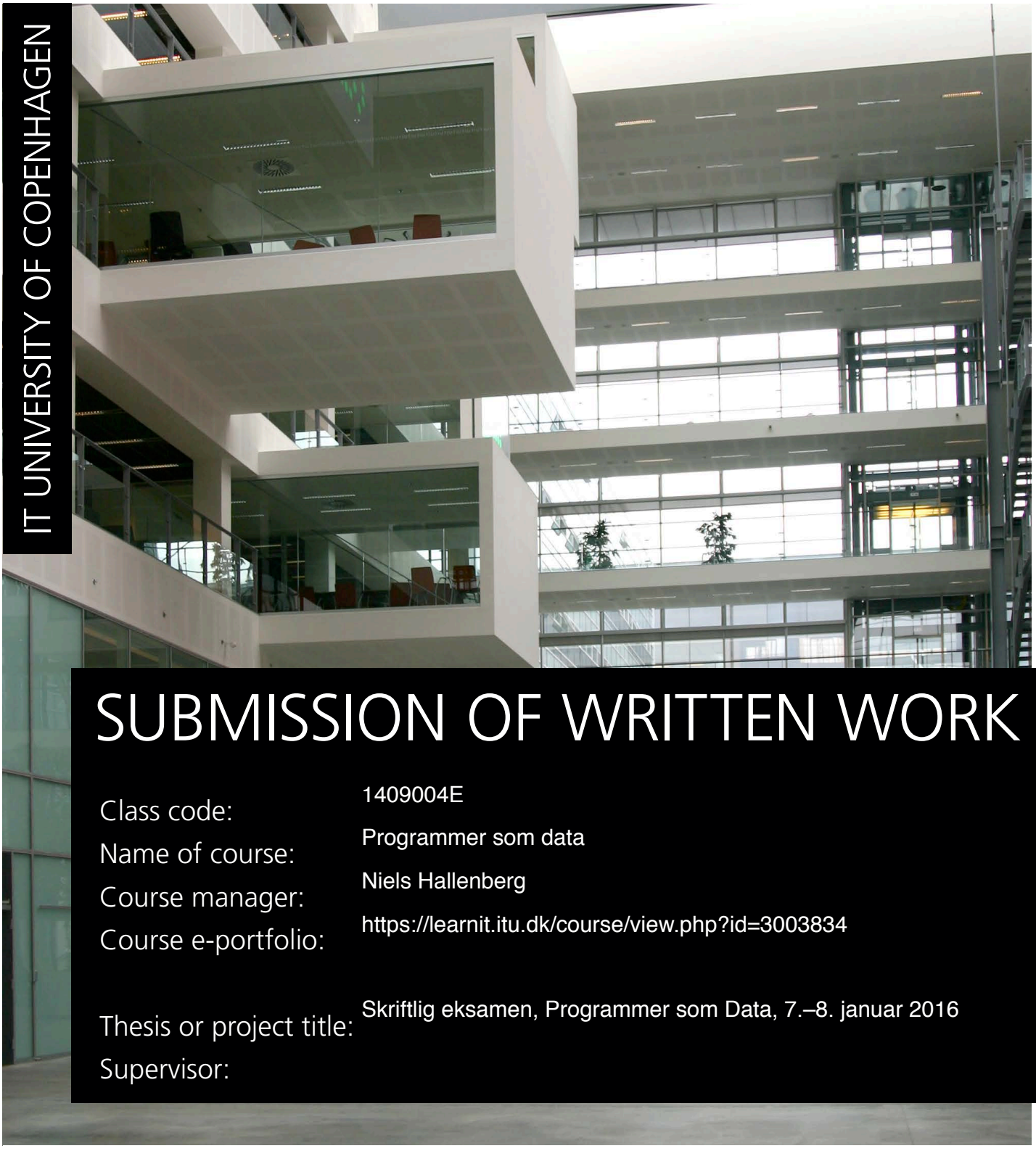
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	1409004E
Name of course:	Programmer som data
Course manager:	Niels Hallenberg
Course e-portfolio:	https://learnit.itu.dk/course/view.php?id=3003834
Thesis or project title:	Skriftlig eksamen, Programmer som Data, 7.–8. januar 2016
Supervisor:	

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. Anders Fischer-Nielsen	06/05-1993	afin@itu.dk
2.		@itu.dk
3.		@itu.dk
4.		@itu.dk
5.		@itu.dk
6.		@itu.dk
7.		@itu.dk

Skriftlig eksamen

Programmer som data

`...` i kodeeksempler beskriver kode, der ikke er ændret, og derfor ikke er inkluderet. `...` er taget med i besvarelsen for at give en fornemmelse for hvor i funktionerne jeg har tilføjet kode.

Jeg erklærer hermed at jeg selv har lavet hele denne eksamensbesvarelse uden hjælp fra andre.

Anders Fischer-Nielsen

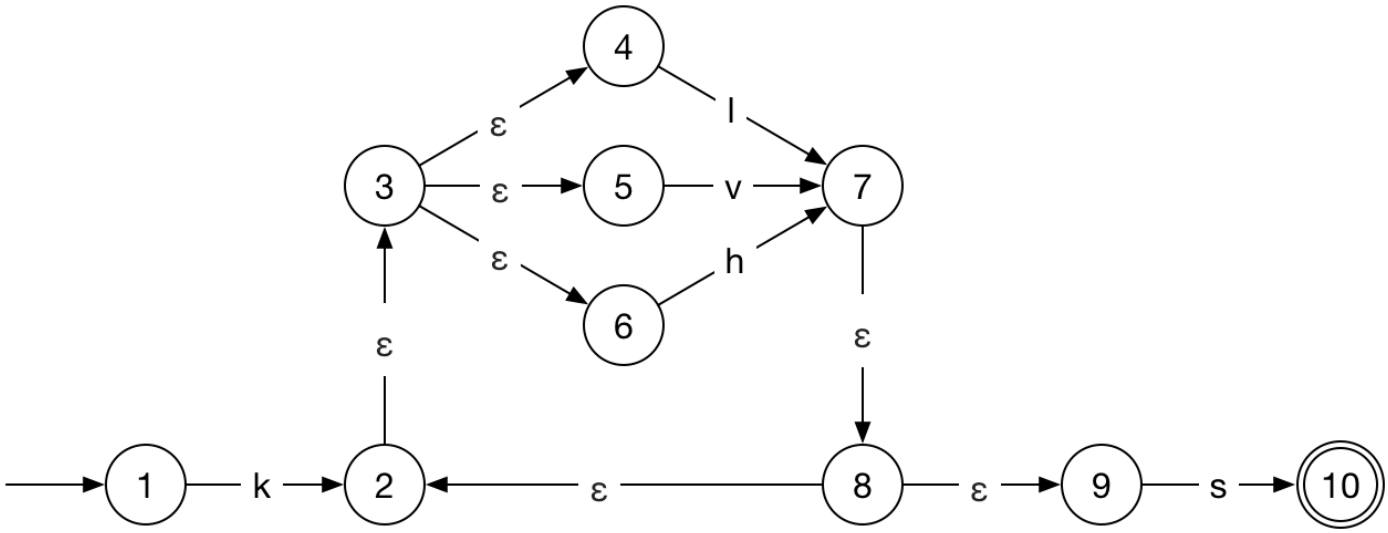
Opgave 1

Opgave 1.1

- kls
- kllllls
- klvhls
- khhhs
- kvs

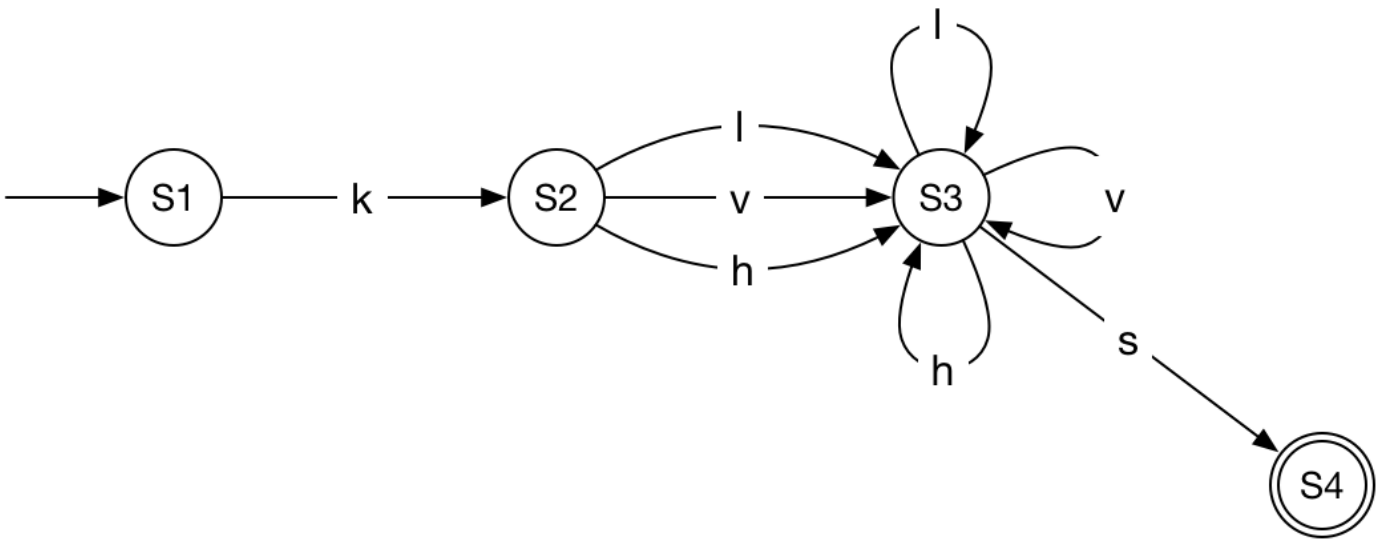
Det regulære udtryk beskriver sproget, hvor ét k altid efterfølges af et eller flere l, v eller h, for til sidst at have ét s.

Opgave 1.2



Jeg har anvendt reglerne beskrevet i Mogensens *"Introduction to Compiler Design"*, beskrevet i kap. 1, til generering af NFA'en.

Opgave 1.3



Jeg har anvendt matrix-metoden, vist til forelæsningerne i faget, til konvertering af NFA til DFA. Matricen jeg endte med kan ses her:

STATE	k	l	v	h	s	NFA
S1	S2	Ø	Ø	Ø	Ø	2
S2	Ø	S3	S3	S3	Ø	2,3,4,5,6
S3	Ø	S3	S3	S3	S4	2,3,4,5,6,7,8,9
S4						10

Opgave 1.4

Jeg er kommet frem til det regulære udtryk `((h|v)*l)+`, der beskriver den angivne mængde strenge.

Ved at teste udtrykket, kan jeg se, at strengene `lv, lh, lvv, lvh, lhv` og `lhh` som ønsket ikke genkendes.

Expression share save flags

`/((h|v)*l)+/g` **27 matches**

```
ØNSKES GENKENDT:
l,    vl,   hl
ll,   lvl,  lhl,  vll,  vlvl,    vlhl,    hll,  hlvl,    hlhl
lll,  llvl, llhl, lvll, lvlvl,   lvlhl,   lhll  lhlvl    lhlhl

ØNSKES IKKE GENKENDT:
lv, lh, lvv, lvh, lhv lhh
```

Opgave 2

Opgave 2.1

`Absyn.fs` er ændret ved at tilføje følgende typer:

```
type expr =
    ...
  | Ref of expr
  | Deref of expr
  | UpdRef of expr * expr
```

Opgave 2.2 & 2.3

`HigherFun.fs` er ændret ved at tilføje følgende type:

```
type value =
  ...
  | RefVal of value ref
```

Hermed kan reference-typer repræsenteres.

Derudover er `eval` ændret:

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    ...
    | Ref e -> RefVal (ref (eval e env))
    | Deref e ->
      let v = eval e env
      match v with
      | RefVal v' -> !v'
      | _         -> failwith "%A is not a reference type." v
    | UpdRef(e1, e2) ->
      let v1 = eval e1 env
      match v1 with
      | RefVal v1' -> let v2 = eval e2 env
                      v1' := v2
                      v2
      | _          -> failwith "%A is not a reference type." e1
```

Hermed muliggøres evaluering af reference-typer.

En `Ref e` evaluerer `e`.

En `Deref e` evaluerer `e`, *pattern matcher* på resultatet for at sikre, at `e` er en reference-type, og returnerer til sidst resultatet, hvis dette er tilfældet. Hvis ikke fejler funktionen.

En `UpdRef (e1, e2)` evaluerer `e1`, *pattern matcher* igen på resultatet for at sikre, at `e1` er en reference-type, evaluerer `e2`, og tildeler resultatet til e1. Til slut returneres `e2`. Hvis `e1` ikke er en reference-type, fejler funktionen.

Opgave 2.4

I eksemplerne der ses nedenfor er det abstrakte syntakstræ efterfulgt af det Micro-ML, det repræsenterer, efterfulgt af resultatet af evalueringen af træet.

```
let ex1 = Let ("x",Ref (CstI 2), Let ("y",Ref (CstI 3), UpdRef (Var "y",Prim ("+",
Deref (Var "x"), Deref (Var "y")))))

let x = 2 in
  let y = 3 in
    y = x + y
  end
end

val it : HigherFun.value = Int 5
```

```
let ex2 = Let ("x", Ref (CstI 1), UpdRef (Var "x", CstI 2))

let x = 1 in
  x = 2
end

val it : HigherFun.value = Int 2
```

```
let ex3 = Let ("y", Ref (CstI 1), UpdRef (Var "y", Deref (Var "y")))

let y = 1 in
  y = y
end

val it : HigherFun.value = Int 1
```

```
let ex4 = Letfun ("deref", "x", Deref(Var "x"),Call (Var "deref",Ref (CstI 2)))

let deref x = !x in f ref 2 end

val it : HigherFun.value = Int 2
```

Opgave 2.5

Følgende ændringer er lavet i `FsLex.fsl`:

```
let keyword s =
    ...
    | "ref"   -> REF
    ...

 let token = parse
    ...
    | '!'             { DEREF }
    | ":="            { ASSIGN }
    ...
```

Følgende ændringer er lavet i `FunPar.fsy`:

```
%token ELSE END FALSE IF IN LET NOT THEN TRUE REF
%token EQ NE GT LT GE LE DEREF ASSIGN

%left EQ NE
%right REF
%right DEREF
...
%right ASSIGN
%nonassoc NOT

Expr:
  ...
  | DEREF Expr         { Deref($2)      }
  | Expr ASSIGN Expr   { UpdRef($1, $3) }

AtExpr:
  ...
  | REF Expr           { Ref $2         }
```

`ref` er oprettet som keyword, `!` og `:=` er oprettet som tokens, hhv. `DEREF` og `ASSIGN`.

Ovenstående er højre-associative, da f.eks `y!x` binder sig til at hente værdien af `x`, `ref` binder sig til værdien på højre side, og `:=` assigner venstre-siden med værdien på højre side.

`DEREF Expr` parses som ovenstående, da en `Expr` godt kan resultere i en reference-type, og parseren derfor ikke kan være strengere.

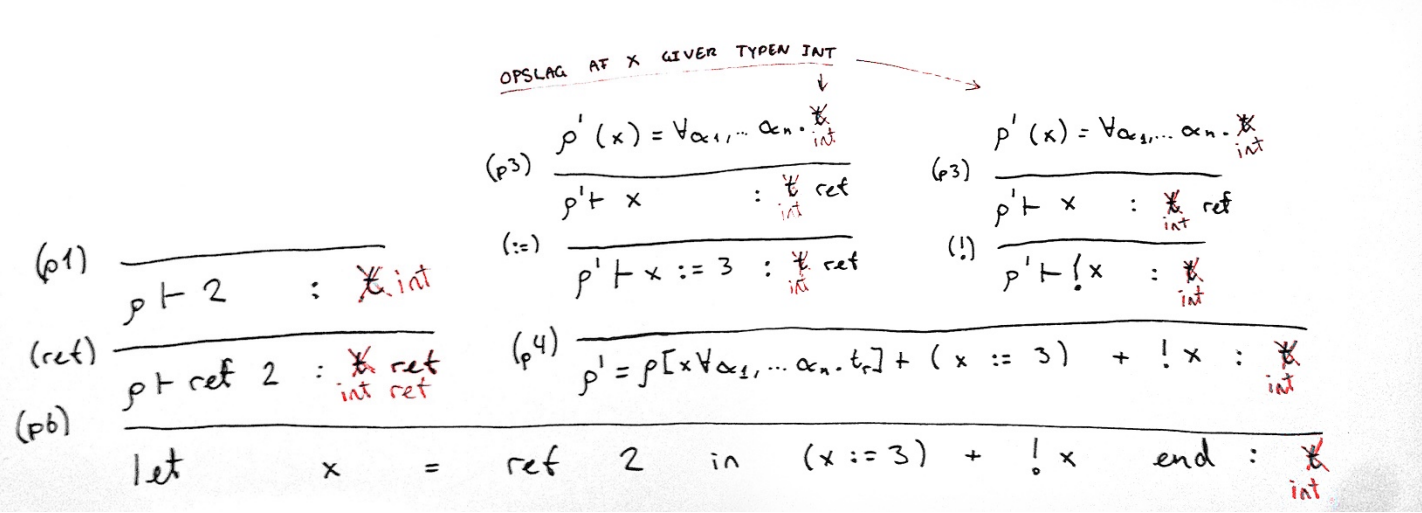
Det samme gælder for `ASSIGN` (dog tjekkes der for lovlige reference-typer i `eval` efterfølgende).

Opgave 2.6

```
let ex1 = fromString "let x = ref 1 in if !x = 1 then x := 2 else 42 end";;
val ex1 : Absyn.expr =
  Let
    ("x",Ref (CstI 1),
     If (Prim ("=",Deref (Var "x"),CstI 1),UpdRef (Var "x",CstI 2),CstI 42))

> run ex1;;
val it : HigherFun.value = Int 2
```

```
let ex2 = fromString "let x = ref 2 in (x := 3) + !x end";;

val ex2 : Absyn.expr =
  Let ("x",Ref (CstI 2),Prim ("+",UpdRef (Var "x",CstI 3),Deref (Var "x")))

> run ex2;;
val it : HigherFun.value = Int 6
```

Opgave 2.7



Håndskrevet typetræ. Jeg er kommet til at skrive typereglen til venstre i stedet for højre, modsat Sestofts PLC.

Rød markerer senere indsættelse af funden type.

Opgave 3

Opgave 3.1

I `Absyn.fs` er følgende tilføjet:

```
and expr =
  ...
  | CstS of string
  ...
```

I `CPar.fsy` er følgende tilføjet:

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

I `Machine.fs` er følgende tilføjet:

```
type instr =
  ...
  | CSTS of string
```

```
...
let CODESETCDR = 31;
let CODECSTS   = 32;
```

```
let makelabenv (addr, labenv) instr =
    match instr with
    ...
    | CSTS s         -> (addr+s.Length+2, labenv)
```

```
let rec emitints getlab instr ints =
    match instr with
    ...
    | CSTS s -> CODECSTS :: (String.length s) :: ((explode s) @ ints)
```

I `Comp.fs` er følgende tilføjet:

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr list =
    match e with
    ...
    | CstS s         -> [CSTS s]
    ...
```

I `listmachine.c` er følgende tilføjet:

```
#define CONSTAG 0
#define STRINGTAG 1
```

```
#define SETCDR 31
#define CSTS 32
```

```
int execcode(int p[], int s[], int iargs[], int iargc, int /* boolean */ trace) {
  ...
    switch (p[pc++]) {
    ...
    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;
    ...
```

Keywords for konstante strenge er tilføjet til lexer og parser, således at disse kan repræsenteres i det abstrakte syntaks-træ. Når labels skal oprettes i `Machine.fs` gøres der plads til strengen ved at lægge den givne strengs længde + 2 til den nuværende adresse. Dette gøres, da der også skal være plads til strengens *header* plus `lenStr` inden strenges indhold. Herefter lægges strengens indhold ind ved at konvertere hver karaktér til en `int`.

Under eksekvering af den generede bytecode, når instruktionen for `CSTS` læses, læses næste program-instruktion fra `p[]`, hvor strengens længde findes. Der allokeres herefter plads til strengen, og strengens længde og karaktererer skrives til `p[]`. Strengens adresse skrives til sidst ind på stakken.

Opgave 3.2

Af den genererede abstrakte syntax

```
Prog
  [Fundec
     (null,"main",[],
      Block
        [Dec (TypD,"s1"); Dec (TypD,"s2");
         Stmt (Expr (Assign (AccVar "s1",CstS "Hi there")));
         Stmt (Expr (Assign (AccVar "s2",CstS "Hi there again")))])]
```

ses det, at der erklæres to dynamiske variable (`TypD`), kaldet `s1` og `s2` , der herefter tildeles hver sin `CstS` vha. `Assign` og `AccVar` . Disser tilgår ti sammen variablerne og assigner strengene til dem.

Alt dette sker i den deklarerede funktion `"main"` (erklæret vha. `Fundec`), hvor deklarationen af `s1` og `s2` efterfølges af to statements (`Stmt`), der indeholder tildelings-syntaksen beskrevet ovenfor.

Med programmet fra opgaven printes

```
The string "Hi there" has now been allocated.
The string "Hi there again" has now been allocated.

Used   0.000 cpu seconds
```

ved kørsel af `.out` -filen med `listmachine` .

Ved at debugge `listmachine.c` ses det, at længden af strengen, samt mængden af ord der skal allokeres i hoben, er korrekt.

```
(lldb)
Process 6659 stopped
...
   219        int lenStr = p[pc++];
   220        int sizeStr = lenStr + 1; // Extra for zero terminating string, \0.
   221        int sizeW = (sizeStr % 4 == 0)?sizeStr/4:(sizeStr/4)+1;
-> 222        sizeW = sizeW + 1; // Extra for string length.
   223        word* strPtr = allocate(STRINGTAG, sizeW, s, sp);
(lldb) p sizeW
(int) $12 = 3
...
(lldb) p p[pc-1] // = lenStr. p[pc-1] da koden lægger én til pc inden.
(int) $15 = 8
(lldb) p (char) p[pc]
(char) $16 = 'H' //Ønsket indhold
(lldb) p (char) p[pc+1]
(char) $17 = 'i'
```

Det ser hermed ud til, at strengen er gemt korrekt i hoben.

Analyse af de genererede bytecode-instruktioner for programmet giver også tiltro til, at programmet er korrekt.

```
LDARGS;
CALL (0,"L1");          //Gå til første funktion, main().
STOP;

Label "L1";             //main() funktion
NIL;                    //Opret NIL variable
NIL;                    //Opret NIL variable
GETBP;
CSTI 0;
ADD;                    //Placering af første variabel (på BP+0)
CSTS "Hi there";        //Opret streng
STI;                    //Gem i første variabel
INCSP -1;
GETBP;
CSTI 1;
ADD;                    //Placering af anden variabel (på BP+1)
CSTS "Hi there again"; //Samme som ovenfor, bare anden streng
STI;                    //Gem streng i anden variabel
INCSP -1;
INCSP -2;               //Smid alt væk (intet returneres)
RET -1
```

Opgave 4

Opgave 4.1

I `CLex.fsl` er følgende pattern match tilføjet:

```
rule Token = parse
  ...
  | ".=="           { DOTEQ }
  | ".!="           { DOTNE }
  | ".>"            { DOTGT }
  | ".<"            { DOTLT }
  | ".>="           { DOTGE }
  | ".<="           { DOTLE }
```

I `CPar.fsy` er følgende tilføjet:

```
ExprNotAccess:
  ...
  | IntervalCheck   { $1    }

IntervalCheck:
    Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }
  | Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }
  | Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }
  | Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }
  | Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }
  | Expr PrimOp Expr PrimOp Expr { Andalso(Prim2($2, $1, $3), Prim2($4, $3, $5)) }

PrimOp:
    DOTEQ                               { "=="              }
  | DOTNE                               { "!="              }
  | DOTGT                               { ">"               }
  | DOTLT                               { "<"               }
  | DOTGE                               { ">="              }
  | DOTLE                               { "<="              }
```

Interval-checks bliver lavet til passende tokens i `CLex.fsl`.

I `CPar.fsy` er typen `IntervalCheck` tilføjet, der tjekker hvorvidt et interval check er til stede.

`PrimOp`-typen er introduceret for at kunne anvende denne direkte i `IntervalCheck`'s `Andalso`-returværdi. Der returneres derfor blot en tekststreng, der angiver `Prim2`-operatoren.

Implementationen tillader derfor ikke en uendelig række af interval check, men kræver at disse er i par af to `PrimOp` med tre `Expr` imellem (dog tillades *nestede* interval check).

Opgave 4.2

```
void main() {
    print -200 .< 2400 .> 3;
    print 2+2 .< 2/2 .== 1;
    print (1 .== 1 .== 2-1) .== 1 .<= 3;
    print (2 .== 2 .== 2) .<= 3 .!= 1;
    print -2 .!= 2 .>= 0;
}
```

Ovenstående tests giver resultatet

```
1 0 1 1 1
```

der er som forventet. Det, at bolske værdier oversættes til integers, er synligt i eksempel tre og fire, hvor den bolske værdi `true` sammenlignes med tallet 1.

`Expr` (som f.eks. `2+2`) oversættes også korrekt inden interval check, som det ses i eksempel to og tre.

Hvis bare én af the to interval check er `false` er hele udtrykket også `false`, som det ses i eksempel 2.

Resultaterne er derfor som forventet.