Programmer som data exam

Albert Ross Johannessen

December 14, 2024

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

1 Test exam 2024

1.1 Opgave 1

Betragt dette regulære udtryk over alfabetet $\{d,',',\}$, hvor d står for decimalt ciffer og ',' er komma:

$$d+','?d* \tag{1}$$

Ved antagelse af, at d svarer til tallene fra 0 til 9 og ',' er et komma, så beskriver det regulære udtryk kommatal.

1.1.1

Udtrykket kan beskrive tal i sættet $\mathbb{R}_0^+$, dette vil altså sige vilkårligt store reelle tal.
Følgende eksempler er inkluderet i sættet:

- 0
- $1,1$
- $9,999999999$
- 20
- $123456,0$
- $2,$

1.1.2

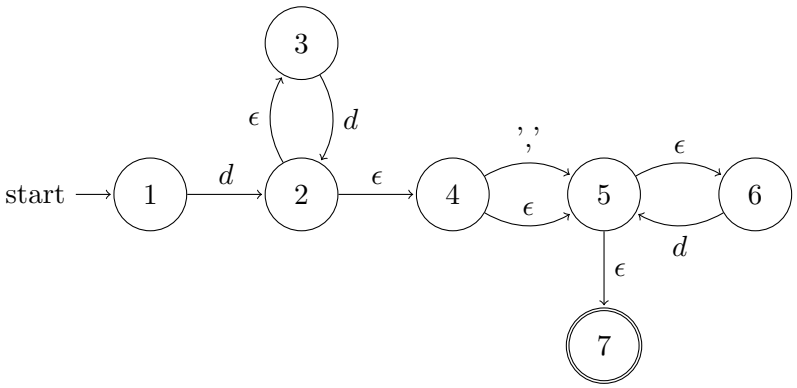

Figure 1: The labeled NFA representing $d+$

1.1.3 Vil tilstandsmaskinen acceptere netop de strenge, som genkendes af det regulære udtryk ovenfor

Ja man kan dele det regulære udtryk op i diskrete dele. Dette kan vi sammenligne med tilstandsmaskinen angivet nedenfor.

Det er åbenlyst at se at disse alle er dele af den tilstandsmaskine angivet.

Tilstandsmaskinen angivet er ikke deterministisk da man kan være i flere knuder på en gang på grund af eksistensen af epsilon kanter.

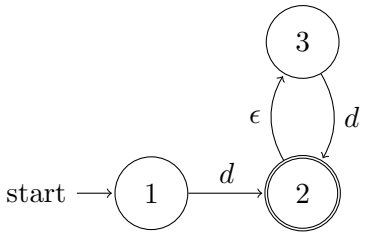

Figure 2: The labeled NFA representing $d+$

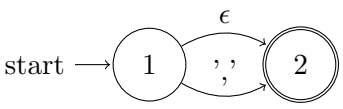

Figure 3: The labeled NFA representing $','?$

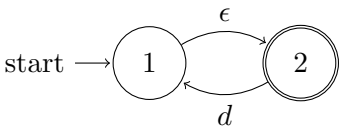

Figure 4: The labeled NFA representing $d*$

1.1.4

$$(d+','?d*)? \tag{2}$$

1.1.5

Løsningen kan findes i kommatal.fsl

1.2 Opgave 2

```
let exam2_1 = Every(Write(FromTo(1, 6)));

let exam2_2 = Every(
  Write(
    Prim("+",
      Prim("*", CstI 10, FromTo(3, 6)),
      FromTo(3,4)
    )
  )
)

let exam2_3 = Every(Write(FromToBy(1, 10, 3)))

let exam2_4 = Every(
  Write(
    Prim("+",
      FromToBy(30, 60, 10),
      FromTo(3,4)
    )
  )
)
```

```
let exam2_5 = Every(
  Write(
      FromToBy(10, 11, 0)
  )
)
```

1.3 Opgave 3

```
type expr =
    | CstI of int
    | CstB of bool
    | Var of string
    | Let of string * expr * expr
    | Prim of string * expr * expr
    | If of expr * expr * expr
    | Letfun of string * string * expr * expr    (* (f, x, fBody, letBody) *)
    | Call of expr * expr
    | Print of expr

let keyword s =
    match s with
    | "else"  -> ELSE
    | "end"   -> END
    | "false" -> CSTBOOL false
    | "if"    -> IF
    | "in"    -> IN
    | "let"   -> LET
    | "not"   -> NOT
    | "then"  -> THEN
    | "true"  -> CSTBOOL true
    | "print" -> PRINT
    | _       -> NAME s

%token ELSE END FALSE IF IN LET NOT THEN TRUE PRINT

Expr:
    AtExpr                              { $1                    }
  | AppExpr                             { $1                    }
  | IF Expr THEN Expr ELSE Expr         { If($2, $4, $6)        }
  | MINUS Expr                          { Prim("-", CstI 0, $2) }
  | Expr PLUS  Expr                     { Prim("+",  $1, $3)    }
  | Expr MINUS Expr                     { Prim("-",  $1, $3)    }
  | Expr TIMES Expr                     { Prim("*",  $1, $3)    }
  | Expr DIV   Expr                     { Prim("/",  $1, $3)    }
  | Expr MOD   Expr                     { Prim("%",  $1, $3)    }
  | Expr EQ    Expr                     { Prim("=",  $1, $3)    }
  | Expr NE    Expr                     { Prim("<>", $1, $3)    }
  | Expr GT    Expr                     { Prim(">",  $1, $3)    }
  | Expr LT    Expr                     { Prim("<",  $1, $3)    }
  | Expr GE    Expr                     { Prim(">=", $1, $3)    }
  | Expr LE    Expr                     { Prim("<=", $1, $3)    }
```

```
  | PRINT      Expr                    { Print($2)              }
;

| Print(ex) ->
let evaluated = eval ex env
printfn "%A" evaluated
evaluated;;
```

2 Januar 2019

2.1 Opgave 3

$$e_6\ \dfrac{e_{10}\ \dfrac{e_1\ \overline{\rho \vdash 32 \Rightarrow 32}}{\rho \vdash \{\text{field1} = 32\} \Rightarrow \{(\text{field1}; 32)\}}\quad \dfrac{e_{10}\ \dfrac{e_3\ \overline{\rho\,(x) = \{(\text{field1}; 32)\}}}{\rho \vdash x \Rightarrow \{(\text{field1}; 32)\}}}{\rho\,[x \mapsto \{(\text{field1}; 32)\}] \vdash \texttt{x.field1} \Rightarrow 32}\ e_{11}}{[] \vdash \texttt{let x= \{field1=32\} in x.field1 end} \Rightarrow 32}$$

2.2 Opgave 5

2.2.1 Opgave 5.1

På toppen af stakken bliver der allokeret 5 elementer til `a`

```
[
---------------main---------------
  4               ret addr
  -999            old bp
  42 42 42 42 42  a[0]-a[4]
  2               ref a
  5               i
------------printArray------------
  125             ret addr
  2               old bp
  2               arg a
  0               i
]
```

2.2.2 Opgave 5.2

Der er en bug i den måde vi håndtere halekald. Da printarray bliver til den sidste funktion kaldt af main så fjerner den alle de lokale variable oprettet af main. Hvilket gør at printarray læser "tilfældige" værdier fra stakken alt efter hvad der ligger på toppen af stakken når der bliver lavet `LDI` kald.

3 December 2019

3.1 Opgave 1

Betragt den ikke-deterministiske endelige automat ("nondeterministic finite automaton", NFA) nedenfor. Det anvendte alfabet er $\{a, b\}$. Der er i alt 5 tilstande, hvor tilstand 5 er den eneste accepttilstand.

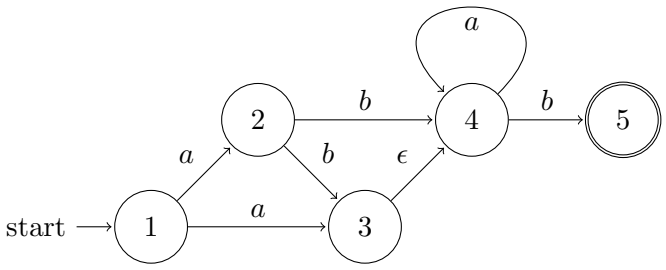

Figure 5: The labeled NFA

3.1.1 Angiv alle årsager til at automaten er ikke-deterministisk

- Fra knude 1 er der 2 a kanter.
- Fra knude 2 er der 2 b kanter.
- Fra knude 3 er der èn ϵ kant.

3.1.2 Giv tre eksempler på strenge der genkendes af automaten

- abb
- abaaaab
- aaaaaab

3.1.3 Giv en uformel beskrivelse af sproget (mængden af all strenge) der beskrives af automaten

- Der er to måder at starte på denne automat.

  1. Ét a og herefter ét b.
  2. Ét eller flere a.

  Herefter er slutningen ens man kan vælge et vilkårligt antal a og herefter **skal** man slutte af på ét b. Kanten fra 2 til 3 er overflødig og det samme er knude 3, hvis man sætter en kant mellem 1 og 4.

3.1.4 Konstruer og tegn en deterministisk endelig automat

Konstruer og tegn en deterministisk endelig automat ("deterministic finite automaton", DFA) der svarer til automaten ovenfor. Husk at angive starttilstand og accepttilstand (e). Du skal enten bruge en systematisk konstruktion svarende til den i forelæsningen eller som i Introduction to Compiler Design (ICD), eller Basics of Compiler Design (BCD), eller forklare hvorfor den resulterende automat er korrekt.
Vi starter med at konstruerer vores subset.

	a	b	NFA State
S_1	$\{2,3,4\}^{S_2}$	$\{\}$	$\{1\}$
S_2	$\{4\}^{S_3}$	$\{3,4,5\}^{S_4}$	$\{2,3,4\}$
S_3	$\{\}^{S_3}$	$\{5\}^{S_5}$	$\{4\}$
S_4	$\{\}^{S_3}$	$\{5\}^{S_5}$	$\{3,4,5\}$
S_5	$\{\}$	$\{\}$	$\{5\}$

Dette giver følgende DFA, vi skal dog verificere at den er minimal.

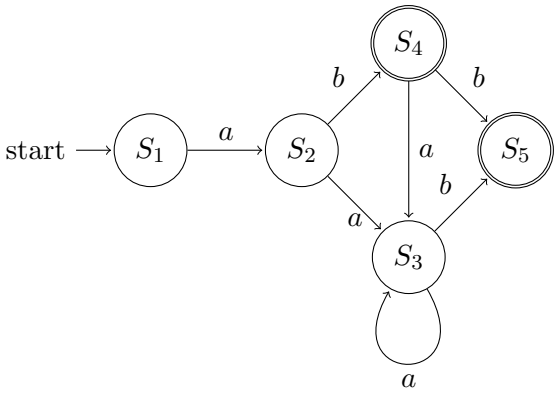

Figure 6: The labeled DFA

Vi starter med at tjekke om grupperne er konsistente.

G_1	$\{S_1, S_2, S_3\}$
G_2	$\{S_4, S_5\}$

G_1	a	b
S_1	$\{G_1\}$	-
S_2	$\{G_1\}$	$\{G_2\}$
S_3	$\{G_1\}$	$\{G_2\}$

G_1 er ikke konsistent så vi opretter 2 nye grupper.

G_2	$\{S_4, S_5\}$
G_3	$\{S_1\}$
G_4	$\{S_2, S_3\}$

G_2	a	b
S_4	$\{G_4\}$	$\{G_2\}$
S_5	-	-

G_3	$\{S_1\}$
G_4	$\{S_2, S_3\}$
G_5	$\{S_4\}$
G_6	$\{S_5\}$

G_3	a	b
S_1	$\{G_4\}$	-

G_4	a	b
S_2	$\{G_4\}$	$\{G_5\}$
S_3	$\{G_4\}$	$\{G_6\}$

G_3	$\{S_1\}$
G_5	$\{S_4\}$
G_6	$\{S_5\}$
G_7	$\{S_2\}$
G_8	$\{S_3\}$

There are only singletons left, this means that we have a minimal DFA.

3.1.5 Angiv et regulært udtryk

Angiv et regulær udtryk der beskriver mængden af strenge over alfabete $\{a, b\}$, som beskrives af automaten ovenfor. Check og forklar at det regulære udtryk også beskriver mængden af strenge for din DFA.

$$ab?a * b \tag{3}$$

3.2 Opgave 2

$$\dfrac{p_1\ \dfrac{}{\rho \vdash 21 : \texttt{int}} \qquad \dfrac{p_3\ \dfrac{\rho(\texttt{x}) = \forall\alpha.\alpha \to \texttt{int}}{\rho \vdash \texttt{x} : \texttt{int}}\ p_4\ \dfrac{p_1\ \dfrac{}{\rho \vdash 2 : \texttt{int}}\ \dfrac{}{\rho \vdash 3 : \texttt{int}}\ p_1}{\rho \vdash \texttt{2+3} : \texttt{int}}\ \dfrac{}{\rho \vdash 40 : \texttt{int}}\ p_1}{\rho\,[f \to \forall\alpha_1 \ldots \alpha_n.t_x \to t_r] \vdash \texttt{x within [2+3, 40]} : \texttt{bool}}\ inCheck}{\rho \vdash \texttt{let x = 23 in x within [2+3, 40] end} : \texttt{bool}}\ p_6$$

3.3 Opgave 3

3.3.1

Givet følgende kode
Kan vi se at værdien 7 angiver stak dybden.

- a - har et lokalt offset på 6 med typen array af ints som er på to elementer, da der også skal være plads til en header bliver dette til 3 elementer der bliver allokeret til.
- pn - har et lokalt offset på 3 med typen array af pointers til ins på længden 1, igen på grund af header skal der allokeres til 2 elementer.
- p - har et lokalt offset på 1 med typen pointer til int og dertil skal kun allokeret ét element.
- n - har et lokalt offset på 0 med typen int, og skal kun have allokeret ét element.
- g - har en absolut adresse på 0 med typen int.

Dette vil altså sige at den næste variabel der skal allokeres til skal starte på position 7.

```
let exVarEnv : varEnv =
  ([
    ("a", (Locvar 6, TypA(TypI, Some 2)));
    ("pn", (Locvar 3, TypA(TypP TypI, Some 1)));
    ("p", (Locvar 1, TypP TypI));
    ("n", (Locvar 0, TypI));
    ("g", (Glovar 0, TypI))
  ], 7)
```

Figure 7

4 2018

4.1 Opgave 1

Betragt den ikke-deterministiske endelige automat (”nondeterministic finite automaton”, NFA) nedenfor. Det anvendte alfabet er {a,b}. Der er i alt 5 tilstande, hvor 5 er den eneste accept-tilstand.

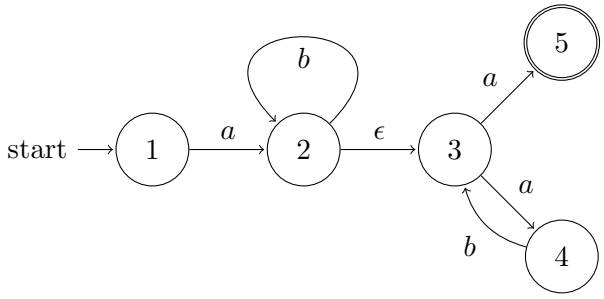

Figure 8: The labeled NFA

4.1.1 Angiv all årsager til at automaten er ikke-deterministisk

1. Der går en epsilon kant fra 2 til 3.
2. Der går 2 a kanter fra 3.

4.1.2 Giv tre eksempler på strenge der genkendes af automaten

1. aa
2. abbba
3. ababa

4.1.3 Giv en uformel beskrivelse af sproget (mængden af alle strenge) der beskrives af automate

Sproget der beskrives af automaten har følgende regler.

- Strenge skal starte og slutte på a
- Det første a kan være efterfulgt af et vilkårligt antal b’er
- I den efterfølgende streng, hvis man vil have mere end ét a så skal alle a’er være sepereret af b’er

4.1.4 Konstruer den tilsvarende DFA

	a	b	NFA State
S_1	$\{2,3\}^{S_2}$	$\{\}$	$\{1\}$
S_2	$\{4,5\}^{S_3}$	$\{\}^{S_2}$	$\{2,3\}$
S_3	$\{\}$	$\{3\}^{S_4}$	$\{4,5\}$
S_4	$\{4,5\}^{S_3}$	$\{\}$	$\{3\}$

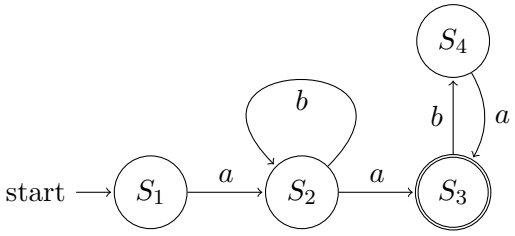

Figure 9: The labeled NFA

G_1	$\{S_1, S_2, S_4\}$
G_2	$\{S_3\}$

G_1	a	b
S_1	G_1	-
S_2	G_2	G_1
S_4	G_2	-

G_2	$\{S_3\}$
G_3	$\{S_1\}$
G_4	$\{S_2\}$
G_5	$\{S_4\}$

Siden der er én gruppe pr knude så er vores DFA så lille som den kan være.

4.1.5 Angiv et regulært udtryk for automaten

Hvis vi kigger på NFA'en som vi får givet i opgave beskrivelsen så kan vi splitte den op.

$$1 \xrightarrow{a} 2 = a$$

$$2 \xrightarrow{b} 2 = b*$$

$$3 \xrightarrow{a} 4 \xrightarrow{b} 3 = (ab)*$$

$$3 \xrightarrow{a} 5 = a$$

Hvis vi sætter dem sammen får vi følgende udtryk.

```
ab*(ab)*a
```

Det samme kan vi gør for DFA'en

$$S_1 \xrightarrow{a} S_2 = a$$
$$S_2 \xrightarrow{b} S_2 = b*$$
$$S_2 \xrightarrow{a} S_3 = a$$
$$S_3 \xrightarrow{b} S_4 \xrightarrow{a} S_3 = (ba)*$$

Hvilket producerer følgende udtryk.

```
ab*a(ba)*
```

Her er det indlysende at se at at det er det samme udtryk men skrevet på en anden måde, der er som udgangspunkt ikke nogen forskel mellem `a(ba)*` og `(ab)*a`.

4.2 Opgave 3

4.2.1 Tegn evalueringstræ

$$\dfrac{e_{10}\ \dfrac{e_6\ \dfrac{e_4\ \dfrac{e_1\ \dfrac{}{\rho \vdash 1 \Rightarrow 1}\quad \dfrac{\rho'(\texttt{Weekend}) = \texttt{Enum}\{\texttt{Sat}_0, \texttt{Sun}_1\}\ \texttt{Sun} = \texttt{Sun}_i\ i = 1}{\rho \vdash \texttt{Weekend.Sun} \Rightarrow 1}\ e_{11}\ v = 1 + 1}{\rho \vdash \texttt{1 + Weekend.Sun} \Rightarrow 2}\quad \dfrac{e_3\ \dfrac{\rho(r) \Rightarrow 2}{\rho \vdash r \Rightarrow 2}\quad \dfrac{}{\rho \vdash 1 \Rightarrow 1}\ e_1\ v = 2 + 1}{\rho[r \mapsto 2] \vdash r + 1 \Rightarrow 3}\ e_4}{\rho' = \rho[\texttt{Weekend} \to \texttt{Enum}\{\texttt{Sat}_0, \texttt{Sun}_1\}] \vdash \texttt{let r = 1 + Weekend.Sun in r + 1 end}}}{[] \vdash \texttt{enum Weekend=Sat | Sun in let r = 1 + Weekend.Sun in r + 1 end end} \Rightarrow 3}$$

4.3 Opgave 4

4.3.1 Opgave 4.1

```
[
-------------main-----------
  -999  old bp
  3     i
--------------f-------------
  24    ret addr PRINTI
  1     old bp
  3     arg
  42    i
]
```

4.3.2 Opgave 4.2

```
1   LDARGS;           Load args
2   CALL (0, "main"); Call Main with 0 args
3   STOP;             Return from Main and end program
4  Label "main";      Label for Main                              [-999]
5   INCSP 1;          Increase stackpointer(sp) with 1 sp = 1 [-999 0]
6   GETBP;            Get base pointer bp = 1                  [-999 0 1]
7   CSTI 3;           Put 3 on the stack                       [-999 0 1 3]
8   STI;              Store indirect                           [-999 3 3]
9   INCSP -1;         Decrease sp with 1                       [-999 3]
10  GETBP;            Get bp = 1                               [-999 3 1]
11  LDI;              Load indirect                            [-999 3 3]
12  CALL (1, "f");    Call function f with 1 argument          [-999 3 13 1 3]
13  PRINTI;           Print integer on top of stack            [-999 3 45]
14  RET 1;            Return and remove vals                   [ ]
15 Label "f";         Label for function f
16  INCSP 1;          Increase sp with 1 sp = 5                [-999 3 13 1 3 0]
17  GETBP;            Get bp = 4                               [-999 3 13 1 3 0 5]
18  CSTI 1;           Push 1 on the stack                      [-999 3 13 1 3 0 5 1]
19  ADD;              Add 1 and 4                              [-999 3 13 1 3 0 6]
20  CSTI 42;          Push 42 on the stack                     [-999 3 13 1 3 0 6 42]
21  STI;              Store indirect                           [-999 3 13 1 3 42 42]
22  INCSP -1;         Decrease sp by 1 sp = 5                  [-999 3 13 1 3 42]
23  GETBP;            Get bp = 4                               [-999 3 13 1 3 42 4]
24  LDI;              Load indirect, gets argument n           [-999 3 13 1 3 42 3]
25  GETBP;            Get bp = 4                               [-999 3 13 1 3 42 3 4]
26  CSTI 1;           Push 1 on the stack                      [-999 3 13 1 3 42 3 4 1]
27  ADD;              Add 4 and 1                              [-999 3 13 1 3 42 3 5]
28  LDI;              Load indirect, gets i                    [-999 3 13 1 3 42 3 42]
29  ADD;              Add 42 and 3                             [-999 3 13 1 3 42 45]
30  RET 2             Return to instruction 13 and remove args[-999 3 45]
```

5 2016

5.1 Opgave 1

Betragt dette regulære udtryk over alfabetet $\{k, l, v, h, s\}$:

$$k(l|v|h|) + s$$

Ved antagelse, at

k svarer til	*kør*
l svarer til	*ligeud*
v svarer til	*venstre*
h svarer til	*højre*
s svarer til	*slut*

så beskriver det regulære udtryk strenge, der symboliserer køreture, eksempelvis turen fra hjem til ITU.

5.1.1 Giv nogle eksempler på strenge der genkendes af det regulære udtryk samt en uformel beskrivelse

Nogle eksempler på strenge der genkendes af det regulære udtryk er.

- kls
- klllllllls
- klvhs
- khhhhs

Sproget beskriver en mængde handlinger man skal tage på en køretur, man kan enten køre ligeud til venstre eller højre, der er følgende krav til en streng.

- Man skal starte med k og slutte med s
- Mellem k og s skal man *mindst* een gang enten dreje eller køre ligeud.

5.1.2 Lav udtrykket om til en NFA

Vi kan starte med at dele udtrykket op i mindre dele og repræsentere dem via en graf visning.

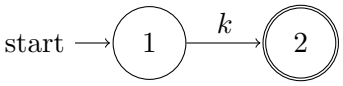

Figure 10: k

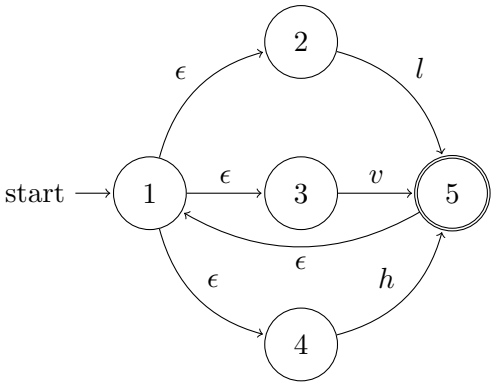

Figure 11: $(l|v|h)+$

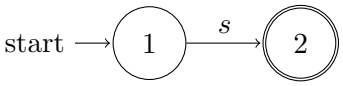

Figure 12: s

Hvis vi sætter figure **?? ?? ??** sammen så får vi følgende graf.

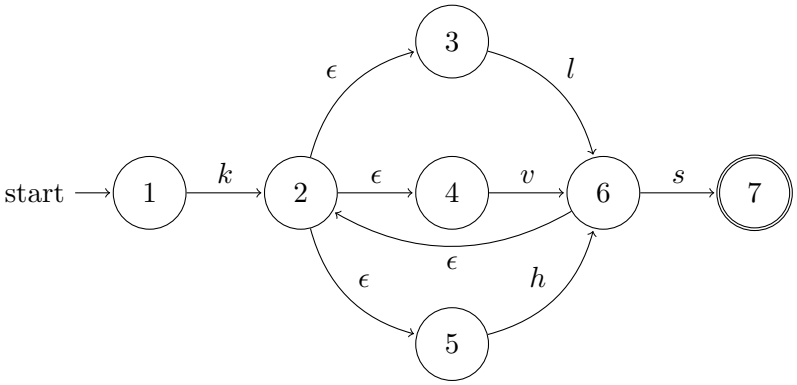

Figure 13: The labeled NFA, representing $k(l|v|h|) + s$

5.1.3 Lav udtrykket om til en DFA

For at konstruerer en DFA fra NFA'en skal vi inddele automaten i subsæt og se hvilke sæt referer til andre sæt. Nu kan vi konstruerer DFA'en. Grunden til at automaten er deterministisk er at

	k	l	v	h	s	NFA state
S_1	$\{2\}^{S_2}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{1\}$
S_2	$\{\}$	$\{6\}^{S_3}$	$\{6\}^{S_3}$	$\{6\}^{S_3}$	$\{\}$	$\{2,3,4,5\}$
S_3	$\{\}$	$\{\}^{S_3}$	$\{\}^{S_3}$	$\{\}^{S_3}$	$\{7\}^{S_4}$	$\{6\}$
S_4	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{7\}$

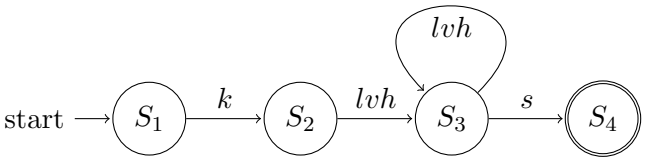

Figure 14: The labeled DFA, representing $k(l|v|h|) + s$

der ikke går to ens kanter fra een knude til to forskellige, der eksisterer heller ikke epsilon kanter i automaten angivet i figur **??**.

5.1.4 Wtf

$$(l + (vl) * (hl)*)|(l * (vl) + (hl)*)|(l * (vl) * (hl)+)$$

5.2 Opgave 2

5.2.1 Udvid typen expr

```
1 type expr =
2     ...
3     | Ref of expr
4     | Deref of expr
5     | UpdRef of expr * expr
```

5.2.2 Udvid typen value

```
1 type value =
2     | Int of int
3     | Closure of string * string * expr * value env
4     | RefVal of value ref
```

5.2.3 Udvid eval i higherfun

Jeg har lavet følgende udvidelse til eval

```
let rec eval (e : expr) (env : value env) : value =
    match e with
    ...
    | Ref e ->
      eval e env |> ref |>  RefVal
    | Deref e ->
      match eval e env with
      | RefVal v -> v.Value
      | _ -> failwith "You can only dereference reference types"
    | UpdRef(e1, e2) ->
      match eval e1 env with
      | RefVal v ->
        let other = eval e2 env
        v.Value <- other
        other
      | _ -> failwith "You you tried to update a non reference
         type"
```

beskrivelsen af koden følger:

- Ref - Vi evaluerer først udtrykket så vi får en værdi som vi kan putte ind i vores ref celle herefter putter vi den ind i en ref celle så vi ikke længere passerer selve værdien rundt men ene reference til den og til sidst pakker vi den i vore "RefVal" type.
- Deref - Da vi kun kan hente værdien fra en reference værdi skal vi tjekke om den rent faktisk har den rigtife type, hvis den har det returnere vi værdien, ellers fejler vi.
- Updref - Samme som "Deref" da vi teknisk set skal "dereference" den så vi kan få værdien, forskellen er at vi herefter opdaterer værdien inde i ref cellen, samt returnerer den nye værdi.

```
let exam2_1 =
  Let("x", Ref(CstI 2),
    Let("y", Var "x",
      Let("z", UpdRef(Var "y", CstI 42),
          Deref(Var "x")
      )
    )
  )
```

Figure 15: Eksempel 1

5.2.4 Udvid lexer og parser

I lexeren "FunLex" bliver der tilføjet 3 nye tokens, hhv. `REF`, `UPD` samt `DEREF`.
I parseren "FunPar" bliver de definerede tokens tilføjet samt vurderet præcedens mæssigt. Her har jeg valgt at give `REF` den laveste præcedens da vi skal kunne definere en værdi med et hvert slags udtryk. Jeg vurderer at `DEREF` skal have højeste præcedens da man typisk binder den direkte til en variabel, og til sidst har jeg valgt at `UPD` skal have samme præcedens som `EQ`.

```
let exam2_2 =
  Let("x", Ref(CstB true),
    Deref(Var "x")
  )
```

Figure 16: Eksempel 2

```
let exam2_3 =
  Let("x", Ref(CstI 10),
    Let("y", Ref(CstI 32),
      Letfun("loop", "i",
        If(
          Prim("=", Deref(Var "x"), CstI 0),
          Deref(Var "y"),
          Let("z",
            UpdRef(Var "y",
              Prim("+",
                Deref(Var "y"),
                CstI 1)
            ),
            Call(Var "loop", UpdRef(Var "x", Prim("-",Deref(Var "
               x"), CstI 1)))
          )
        ),
        Call(Var "loop", Deref(Var "x"))
      )
    )
  )
```

Figure 17: Eksempel 3

```
let exam2_4 =
  Let("x", Ref(CstI 1),
    Let("y", Ref(CstI 1),
      UpdRef(Var "x", Prim("+", Deref(Var "x"), Deref(Var "y")))
    )
  )
```

Figure 18: Eksempel 4

5.2.5 Omskriv eksemplerne

```
{
    let keyword s =
        match s with
        ...
        | "ref"  -> REF
        ...
}

    rule Token = parse
      ...
      | ":="            { UPD }
      | '!'             { DEREF }
      ...
```

Figure 19: Funlex opdateringer

```
%token REF UPD DEREF
%left REF               // lowest precedence
%left ELSE
%left EQ NE UPD
%left GT LT GE LE
%left PLUS MINUS
%left TIMES DIV MOD
%nonassoc NOT DEREF          // highest precedence

Expr:
  ...
  | DEREF Expr     { Deref($2)       }
  | REF Expr       { Ref($2)         }
  | Expr UPD Expr  { UpdRef($1, $3) }
;
```

Figure 20: Funpar opdateringer

```
let x = ref 2 in
  let y = x in
    let z = y := 42 in
      !x
    end
  end
end
```

Figure 21: Implementation af figur **??**

```
let x = ref 2 in
    (x := 3) + !x
end
```

Figure 22: Implementation af figur **??**

```
let x = ref 10 in
  let y = ref 32 in
    let loop i =
        if !x = 0 then !y
        else
          let z = y := !y+1 in
            loop (x:=!x-1)
          end
      in loop (!x)
    end
  end
end
```

Figure 23: Implementation af figur **??**

```
let x = ref 1 in
  let y = ref 1 in
    x := !x+!y
  end
end
```

Figure 24: Implementation af figur **??**

5.2.6 Lav typeinferens for udtryk

$$t_6\ \dfrac{ref\ \dfrac{\rho \vdash \texttt{2} : \texttt{int}}{\rho \vdash \texttt{ref 2} : \texttt{int ref}} \qquad \dfrac{(\texttt{:=})\ \dfrac{t_3\ \dfrac{\rho\,(x) = \texttt{int ref}}{\rho \vdash x : \texttt{int ref}} \quad \dfrac{}{\rho \vdash \texttt{3} : \texttt{int}}\ t_1}{\rho \vdash \texttt{x := 3} : \texttt{int}} \quad \dfrac{\dfrac{\rho\,(x) = \texttt{int ref}}{\rho \vdash x : \texttt{int ref}}\ t_3}{\rho \vdash \texttt{!x} : \texttt{int}}\ (\texttt{!})}{\rho\,[x \mapsto \texttt{int ref}] \vdash \texttt{(x := 3) + !x} : \texttt{int}}\ t_4}{\rho \vdash \texttt{let x = ref 2 in (x := 3) + !x end} : \texttt{int}}$$

Figure 25: Typeinferens af udtrykket `let x = ref 2 in (x := 3) + !x end`

5.3 Opgave 3

5.3.1 Vis de modifikationer du har lavet

```
and expr =
  | Access of access
  | Assign of access * expr
  | Addr of access
  | CstI of int
  | CstS of string
  | CstN
  | Prim1 of string * expr
  | Prim2 of string * expr * expr
  | Andalso of expr * expr
  | Orelse of expr * expr
  | Call of string * expr list
```

Figure 26: Absyn.fs

```
AtExprNotAccess:
    Const                                 { CstI $1                }
  | CSTSTRING { CstS $1 }
  | LPAR ExprNotAccess RPAR               { $2                     }
  | AMP Access                            { Addr $2                }
  | NIL                                   { CstN                   }
  | CONS LPAR Expr COMMA Expr RPAR        { Prim2("cons",$3,$5)    }
  | CAR LPAR Expr RPAR                    { Prim1("car", $3)       }
  | CDR LPAR Expr RPAR                    { Prim1("cdr", $3)       }
  | SETCAR LPAR Expr COMMA Expr RPAR      { Prim2("setcar",$3,$5) }
  | SETCDR LPAR Expr COMMA Expr RPAR      { Prim2("setcdr",$3,$5) }
;
```

Figure 27: CPar.fsy

```
and cExpr (e : expr) (varEnv : varEnv) (funEnv : funEnv) : instr
    list =
    ...
    | Orelse(e1, e2) ->
      let labend  = newLabel()
      let labtrue = newLabel()
      cExpr e1 varEnv funEnv
      @ [IFNZRO labtrue]
      @ cExpr e2 varEnv funEnv
      @ [GOTO labend; Label labtrue; CSTI 1; Label labend]
    | Call(f, es) -> callfun f es varEnv funEnv
    | CstS s -> [CSTS s]
```

Figure 28: Comp.fs

```
1  type instr =
2    | Label of label
3    ...
4    | CSTS of string
5
6  let CODECSTI    = 0
7  ...
8  let CODECSTS = 32;
9
10 let makelabenv (addr, labenv) instr =
11     match instr with
12     | Label lab       -> (addr, (lab, addr) :: labenv)
13     ...
14     | CSTS s -> (addr+1+(Seq.length s), labenv)
15
16 let rec emitints getlab instr ints =
17     match instr with
18     | Label lab       -> ints
19     ...
20     | CSTS s -> CODECSTS :: (String.length s) :: (s |> Seq.map (int) |>
           Seq.toList)  ints
```

Figure 29: Machine.fs

I de forrige udklip kan alle ændringer jeg har lavet ses. Jeg har fulgt opskriften til punkt og prikke, ud over at jeg har lavet min egen version af explode, men som gør det præcis samme. Jeg har også lave en ændring til `makelabenv` som håndtere at placere labels i koden således at `GOTO` er understøttet, her tager jeg den nuværende addresse lægger een til for streng længden og herefter hele strengens længde til.

5.3.2 Vis med eksempelprogrammat at strenge kan oprettes

I den abstrakte syntax bliver der først oprettet en block som indeholde main, herinde bliver der oprettet to dynamiske variable; s1 og s2; Disse bliver begge assignet til `CstS`.

```
1 int execcode(word p[], word s[], word iargs[], int iargc, int /*
    boolean */ trace) {
2   word bp = -999;// Base pointer, for local variable access
3   word sp = -1;// Stack top pointer
4   word pc = 0;// Program counter: next instruction
5   for (;;) {
6     if (trace)
7       printStackAndPc(s, bp, sp, p, pc);
8     switch (p[pc++]) {
9     case CSTS:{
10      int lenStr = p[pc++];
11      int sizeStr = lenStr+1;
12      int sizeW = (sizeStr%4 == 0) ? sizeStr/4: (sizeStr/4)+1;
13      sizeW += 1;
14      word * strPtr = allocate(STRINGTAG, sizeW, s, sp);
15      s[++sp] = (int)strPtr;
16      strPtr[1] = lenStr;
17      char* toPtr = (char*) (strPtr+2);
18      for (int i = 0; i<lenStr; i++)
19        toPtr[i] = (char) p[pc++];
20      toPtr[lenStr] = '\0';
21      printf("The string "\%s" has now been allocated.\n", toPtr)
          ;
22    }break;
23    ...
24    default:
25      printf("Illegal instruction " WORD_FMT " at address "
          WORD_FMT "\n",
26              p[pc - 1], pc - 1);
27      return -1;
28    }
29  }
30 }
```

Figure 30: listmachine.c

5.4 Opgave 4

5.4.1 Udvid micro-c med interval check

```
./listmachine ../../examprog.out
The string "Hi there" has now been allocated.
The string "Hi there again" has now been allocated.

Used 0 cpu milli-seconds
```

```
1 rule Token = parse
2   ...
3   | '.' { DOT }
4   ...
5   | _                { failwith "Lexer error: illegal symbol" }
```

Figure 31: CLex.fsl

```
1 ...
2 %token CHAR ELSE IF INT NULL PRINT PRINTLN RETURN VOID WHILE DOT
3 ...
4 %right ASSIGN             /* lowest precedence */
5 ...
6 %left EQ NE DOT
7 ...
8 %nonassoc LBRACK          /* highest precedence  */
9 ...
10 BoolOp:
11     GE                                   { ">="                }
12   | LE                                   { "<="                }
13   | GT                                   { ">"                 }
14   | LT                                   { "<"                 }
15   | NE                                   { "!="                }
16   | EQ                                   { "=="                }
17 ;
18
19 ExprNotAccess:
20     AtExprNotAccess                      { $1                  }
21   ...
22   | Expr DOT BoolOp Expr DOT BoolOp Expr{ Andalso(Prim2($3, $1,
       $4), Prim2($6, $4, $7))}
23   ...
24   | Expr SEQOR  Expr                     { Orelse($1, $3)      }
25 ;
```

Figure 32: CPar.fsy

I ændringerne ovenfor kan det ses at der kun er tilføjet een ny token og det er `DOT`, dette er så vi kan understøtte i lexeren at den møder et punktum. I vores parser som kan ses i figur **??**, så er der sket lidt mere.

Der er blevet oprettet een nye grammer, `BoolOp`. `BoolOp` er til for at matche med vores understøttede boolske operationer. Der er også blevet tilføjet en ny regel til `ExprNotAccess` som

sørger for at matche den nye funktionalitet, hvilket den gør ved blot at tjekke som der er 3 expressions i streg, herefter smider den dem ind i en `Andalso`.

5.4.2 Udvid micro-c programmet med flere eksempler

```
1  void main(){
2      print 2 .< 3 .< 4;
3      print 3 .< 2 .== 2;
4      print 3 .> 2 .== 2;
5      print (3 .> 2 .== 2) == (3 .> 1 .== 1);
6      print (3 .> 2 .== 2) == 1;
7      // udvidelser opgave 4.2
8      print 2 .< 3 .<= 4;
9      print 2 .> 1 .>= 4;
10     print (2 .>= 2 .>= 4) .== (2 .>= 2 .>= 4) .== (2 .>= 2 .>= 4)
           ;
11     print (2 .>= 2 .>= 4) .!= (2 .<= 2 .<= 4) .== (2 .>= 2 .>= 4)
           ;
12     print (2 .>= 2 .>= 4) .!= (2 .<= 2 .<= 4) .!= (2 .>= 2 .>= 4)
            == (2 .>= 2 .>= 4) .!= (2 .<= 2 .<= 4) .!= (2 .>= 2 .>= 4)
           ;
13 }
```

Figure 33: Det udvidede program

Her testes der flere ting både at alle operatorer er under og at de spiller sammen med sig selv samt eksiterende funktionalitet i koden. Resultaterne er præcis som forventet, dette kan verificeres ved at håndkøre resultaterne hvor man vil finde at svarene stemmer overens med det forventede resultat.