Programmer som data exam

Albert Ross Johannessen

December 12, 2024

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

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.

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.

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.

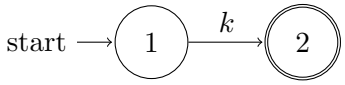

Figure 1: k

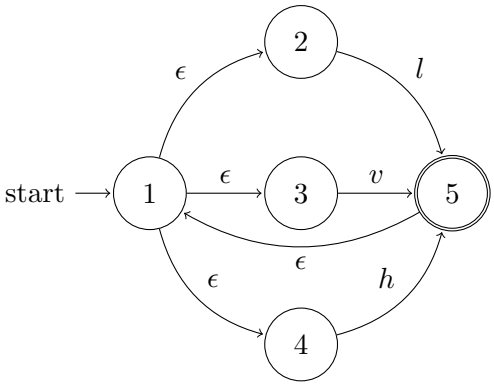

Figure 2: $(l|v|h)+$

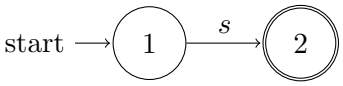

Figure 3: s

Hvis vi sætter figure 1 2 3 sammen så får vi følgende graf.

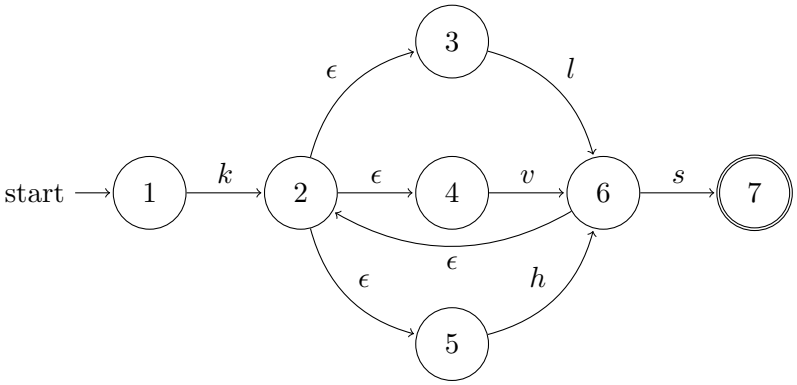

Figure 4: The labeled NFA, representing $k(l|v|h|) + s$

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	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{7\}$

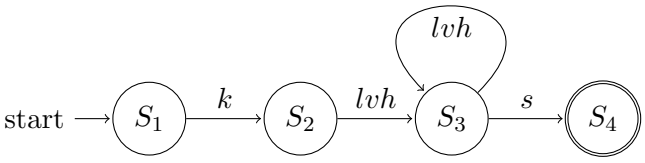

Figure 5: The labeled DFA, representing $k(l|v|h|) + s$

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

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

2 Opgave 2

2.1 Udvid typen expr

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

2.2 Udvid typen value

```
type value =
    | Int of int
    | Closure of string * string * expr * value env
    | RefVal of value ref
```

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")
      )
    )
  )

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

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

2.4 Udvid lexer og parser

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

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

%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) }
```

;

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`.

2.5 Omskriv eksemplerne

```
let x = ref 1 in
    if !x = 1 then x := 2
    else 42
end

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

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

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