Bachelor PRD **IT University, E2011**

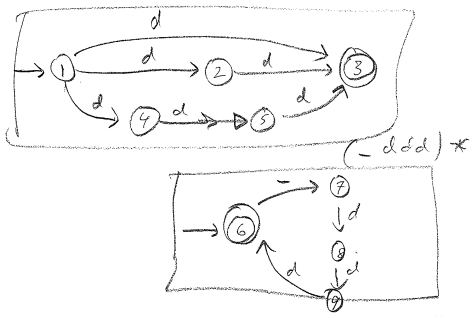
Løsningsforslag
Skriftlig eksamen
9. januar 2012

Version 1, 2012-01-09

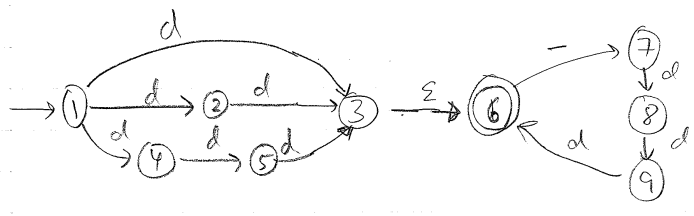
Spørgsmål 1

Spørgsmål 1.1

Først laver vi indlysende korrekt NFAer for hver af de to dele `(ddd|ddd)|` og `(_ddd)*` af det givne regulære udtryk:

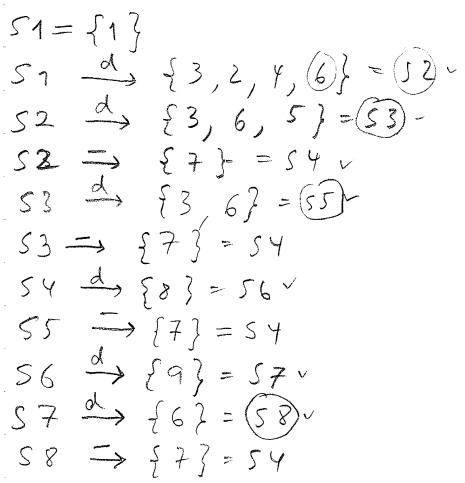


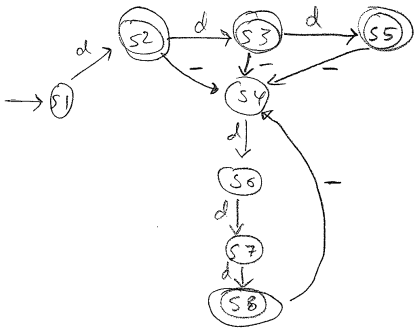
Så sætter vi de to dele sammen med en epsilon-transition (og tilstand 3 er ikke længere) accepttilstand:



Spørgsmål 1.2

De sædvanlige udregninger startende fra S1 foretages. Resultatet er klart en DFA: Ingen epsilon-transitioner og distinkte symboler på forskellige transitioner ud af samme tilstand. DFAen ser også plausibel ud: Med fire accepttilstande S2, S3, S5, og S6 accepterer den sekvenser der består af én, to eller tre cifre, eller som begynder på den måde og ender med en understreg og tre cifre.





Spørgsmål 1.3

Det regulære udtryk kunne være dette, der klart tillader 1 eller flere cifre, men ikke tillader to understreger lige efter hinanden og ikke tillader at tallet begynder eller slutter med et ciffer. Med andre ord, der er mindst ét ciffer på hver side af enhver understreg:

```
[0-9](_?[0-9])*
```

Spørgsmål 2

Spørgsmål 2.1

For at reducere layoutbøvl er `inttree` forkortet til `itr` og $(empty)$ udeladt på alle anvendelser af $(empty)$ reglen:

$$\dfrac{\dfrac{\dfrac{}{\rho \vdash \texttt{11} : \texttt{int}}\,(1) \quad \dfrac{}{\rho \vdash \texttt{10} : \texttt{int}}\,(1)}{\rho \vdash \texttt{11} + \texttt{10} : \texttt{int}}\,(4) \quad \dfrac{}{\rho \vdash \texttt{Empty} : \texttt{itr}} \quad \dfrac{\dfrac{}{\rho \vdash \texttt{42} : \texttt{int}}\,(1) \quad \dfrac{}{\rho \vdash \texttt{Empty} : \texttt{itr}} \quad \dfrac{}{\rho \vdash \texttt{Empty} : \texttt{itr}}}{\rho \vdash \texttt{Node}(\texttt{42}, \texttt{Empty}, \texttt{Empty}) : \texttt{itr}}\,(node)}{\rho \vdash \texttt{Node}(\texttt{11} + \texttt{10}, \texttt{Empty}, \texttt{Node}(\texttt{42}, \texttt{Empty}, \texttt{Empty})) : \texttt{itr}}\,(node)$$

Spørgsmål 2.2

Regel (2) er korrekt. Den forlanger at udtrykket e, som der matches på, har type `inttree`; den forlanger at begge grene har samme type t som så er typen af hele `match`-udtrykket; den forlanger at gren e_1 har denne type yderligere forudsætninger; og den tillader gren e_2 at referere til variablene i og x_1 og x_2 som bindes af matchet mod mønstret `Node(`i`,` x_1`,` x_2`)` og den tillader at antage at i har type `int` og x_1 og x_2 har type `inttree` i e_2.

Regel (1) er forkert fordi den forlanger at første gren e_1 har type `inttree`, en unødvendig restriktion, og hvad der er værre, alligevel tillader hele `match`-udtrykket at have en anden type t.

Regel (3) er forkert fordi den tillader anden gren e_2 at antage at x_1, der jo er bundet til en `inttree`-værdi, har type `int`.

Regel (4) er forkert fordi den forlanger at begge grene e_1 og e_2 har type `inttree`, en unødvendig restriktion, og hvad der er værre, tillader e_2 at antage at x_1 og x_2, der jo er bundet til `inttree`-værdier, at have en urelateret type t.

Regel (5) er forkert fordi den kræver at udtrykket e der skal matches på er en `int`, hvilket ikke giver nogen mening, når den skal have formen `Empty` eller `Node(...)`.

Spørgsmål 2.3

Denne regel er grundlæggende en simplere udgave af regel (2) fra opgave 2.2:

$$\dfrac{\rho \vdash e_r : \texttt{inttree} \qquad \rho[i \mapsto \texttt{int}, x_1 \mapsto \texttt{inttree}, x_2 \mapsto \texttt{inttree}] \vdash e_b : t}{\rho \vdash \texttt{let Node}(i, x_1, x_2) = e_r \texttt{ in } e_b : t}\,(lettree)$$

Spørgsmål 3

Spørgsmål 3.1 og 3.2

Her er parserspecifikationens regeldel (til venstre) og de semantiske regler (til højre), i én omgang:

```
Main:
    States Transitions Payments EOF    { ($1, $2, $3) }
;
States:
     STATES                            { [] }
   | STATES StateSeq                   { $2 }
;
StateSeq:
     NAME                               { [$1] }
   | NAME COMMA StateSeq                { $1 :: $3 }
;
Transitions:
    TRANSITIONS TransitionSeq           { $2 }
;
TransitionSeq:
    /* empty */                         { [] }
   | Transition TransitionSeq           { $1 :: $2 }
;
Transition:
   NAME COLON NAME ARROW NAME           { ($1, ($3, $5)) }
;
Payments:
    PAYMENTS PaymentSeq                 { $2 }
;
PaymentSeq:
    /* empty */                         { [] }
   | Payment PaymentSeq                 { $1 :: $2 }
;
Payment:
   | WHILE NAME PayReceive AMOUNT PER TimeUnit  { Stream($2, $3, $4, $6) }
   | UPON NAME PayReceive AMOUNT                { Lumpsum($2, $3, $4)    }
;
PayReceive:
     PAY                                { Pay     }
   | RECEIVE                            { Receive }
;
TimeUnit:
     MONTH                              { PerMonth }
   | YEAR                               { PerYear  }
;
```

Spørgsmål 3.3 og 3.4

Her er lexerspecifikationens regeldel og token-genererende del, hvor tal skal have tusindtalseparator, i ét:

```
rule Token = parse
  | ...
  | (['0'-'9']|['0'-'9']['0'-'9']|['0'-'9']['0'-'9']['0'-'9'])('_'['0'-'9']['0'-'9']['0'-'9'])*
                    { AMOUNT (System.Int32.Parse (stripUnderscores (lexemeAsString lexbuf))) }
  | ...
```

Den token-genererende del (på højresiden) benytter denne hjælpefunktion til at fjerne understregerne:

```
let rec strip (s: string) i =
    if i >= s.Length then
        ""
    else if s.[i] = '_' then
        strip s (i+1)
    else
        string (s.[i]) + strip s (i+1)

let stripUnderscores s = strip s 0
```

Spørgsmål 4

Spørgsmål 4.1

Nogle forslag til yderligere bytekodeforbedringer:

- Kode af formen `[CSTI i1; CSTI i2; ADD]` kunne forbedres til `[CSTI i3]` hvor `i3` er summen af `i1` og `i2`. Tilsvarede for `SUB` og `MUL`; med `DIV` skal man passe på division med 0, altså først tjekke om `i2` er 0.
- Kode af formen `[SWAP; SWAP]` kan erstattes med `[]`, dvs. fjernes.
- Kode af formen `[GOTO L1; L1: ...]` kan erstattes med `[L1: ...]`. Tilsvarende hvis der står `IFZERO L1` og `IFNZRO L1` i stedet for `GOTO L1`.

Spørgsmål 4.2

Denne `simplify`-funktion implementerer vist alle forenklingerne fra figur 3, samt nogle af dem beskrevet i svaret til 4.1. De to sidste regler tager hensyn til at (1) det kan være der ikke er noget pattern der matcher, og (2) det kan være at koden ikke er længere:

```
let rec simplify code =
    match code with
        | CSTI 0 :: EQ :: rest -> NOT :: simplify rest
        | CSTI 0 :: ADD :: rest -> simplify rest
        | CSTI 0 :: SUB :: rest -> simplify rest
        | CSTI 0 :: NOT :: rest -> simplify (CSTI 1 :: rest)
        | CSTI _ :: NOT :: rest -> simplify (CSTI 0 :: rest)
        | CSTI 1 :: MUL :: rest -> simplify rest
        | CSTI 1 :: DIV :: rest -> simplify rest
        | CSTI 0 :: IFZERO lab :: rest -> simplify (GOTO lab :: rest)
        | CSTI _ :: IFZERO lab :: rest -> simplify rest
        | CSTI 0 :: IFNZRO lab :: rest -> simplify rest
        | CSTI _ :: IFNZRO lab :: rest -> simplify (GOTO lab :: rest)
        | NOT :: NOT :: rest -> simplify rest
        | NOT :: IFZERO lab :: rest -> simplify (IFNZRO lab :: rest)
        | NOT :: IFNZRO lab :: rest -> simplify (IFZERO lab :: rest)
        | INCSP 0 :: rest -> simplify rest
        | INCSP m1 :: INCSP m2 :: rest -> simplify (INCSP (m1+m2) :: rest)
        | INCSP m1 :: RET m2 :: rest -> simplify (RET (m2-m1) :: rest)
        | SWAP :: SWAP :: rest -> simplify rest  (* opgave 4.1 herfra *)
        | DUP :: SWAP :: rest -> simplify (DUP :: rest)
        | CSTI i1 :: CSTI i2 :: ADD :: rest -> simplify (CSTI (i1+i2) :: rest)
        | CSTI i1 :: CSTI i2 :: SUB :: rest -> simplify (CSTI (i1-i2) :: rest)
        | CSTI i1 :: CSTI i2 :: MUL :: rest -> simplify (CSTI (i1*i2) :: rest)
        | ins :: rest -> ins :: simplify rest
        | [] -> []
```

Spørgsmål 4.3

Denne funktion forenkler iterativt indtil `simplify` ikke ændrer koden mere. Dette virker fordi der i F# (ligesom i Scala case classes) er defineret en meningsfuld lighedsoperation (=) på lister og datatyper:

```
let rec simplifyAll code =
    let simpler = simplify code
    if simpler=code then code else simplifyAll simpler
```

Spørgsmål 4.4

Hjælpefunktionen `discardDead` (der meget ligner en fra PLCSD) fjerner alle instruktioner fra begyndelsen af en bytekodesekvens indtil den møder en label, og returnerer så resten, inklusive labelen.

Hjælpefunktionen benyttes af `removeDead`, der leder efter en `GOTO` eller `RET` og så fjerner død kode efter denne.

```
let rec discardDead code =
    match code with
        | Label lab :: rest -> code
        | _ :: rest -> discardDead rest
        | [] -> []

let rec removeDead code =
    match code with
        | GOTO lab :: rest -> GOTO lab :: discardDead rest
        | RET m :: rest -> RET m :: discardDead rest
        | ins :: rest -> ins :: removeDead rest
        | [] -> []
```

Læg mærke til at denne løsning ikke fjerner al død kode, for `removeDead` standser efter den første `GOTO` eller `RET`. For at fjerne al død kode i ét gennemløb, kunne vi kalde `removeDead` på resultatet af `discardDead` i de to første grene af `removeDead`.

Spørgsmål 4.5

Dette gøres mest elegant ved at lave en funktion som matcher på en bytekodeinstruktion for at se hvilke labels, om nogen, instruktionen bidrager med, og så bruge `List.fold` til at gennemløbe alle instruktioner:

```
let addLabel set ins =
    match ins with
        | GOTO lab -> lab :: set
        | IFZERO lab -> lab :: set
        | IFNZRO lab -> lab :: set
        | _ -> set

let usedLabels code = List.fold addLabel [] code
```

Spørgsmål 4.6

Her bruger vi først funktionen `usedLabels` fra opgave 4.5 til at finde ud af hvilke labels der overhovedet bliver hoppet til.

Dernæst defineres en lokal funktion `used(lab)` der returnerer `true` hvis label `lab` findes i denne liste af benyttede labels.

Så defineres endnu en funktion `keepInstruction(ins)` der afgør om en given instruktion `ins` skal bevares. Det skal den hvis den ikke er en label-instruktion, eller hvis den er en label-instruktion for en label der er i brug:

```
let removeUnusedLabels code =
    let labels = usedLabels code
    let used lab = List.exists (function x -> x=lab) labels
    let keepInstruction ins =
        match ins with
            | Label lab -> used lab
            | _ -> true
    List.filter keepInstruction code
```

Bemærk at ved at iterere en kombination af `removeDead` og `removeUnusedLabels` over en kodesekvens kan man rydde grundigere op end de kan hver for sig.