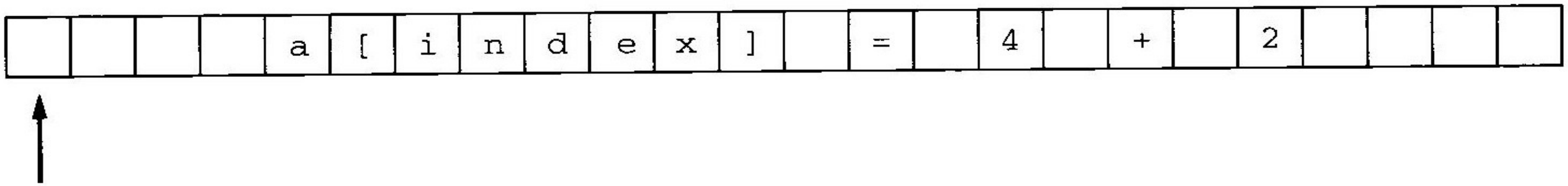
Scanning - I   Kap. 2

- Hovedmål
  - Gå ut fra en beskrivelse av de enkelte tokens, og hvordan de skal deles opp i klasser
  - Lage et program (funksjon, prosedyre, metode) som leverer ett og ett token, med all nødvendig informasjon
  - Tokenet skal angis som en hovedklassifikasjon, med tilhørende attributter
- Vanlige regler
  - Hvert token skal gjøres så langt som mulig innenfor beskrivelsen
  - Dersom et token kan passe med flere beskrivelser må det finnes regler om valg:
    - Typisk: Kan det være både **nøkkelord** og **navn**, så skal det ansees som nøkkelord
  - **Blank**, **linjeskift**, **TAB** og kommentar angir skille mellom tokens, men skal forøvrig fjernes ("white space")

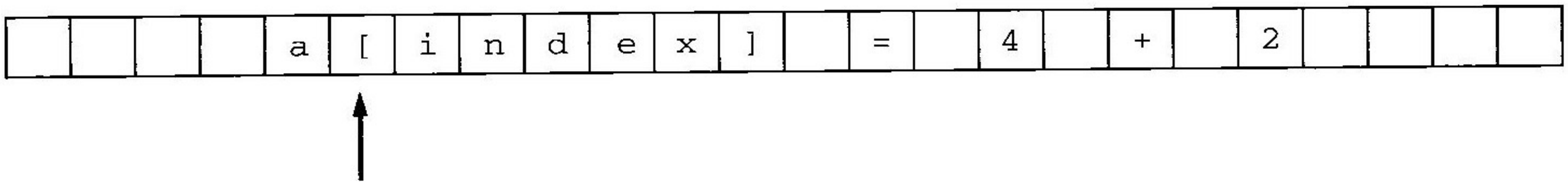
Hva scanneren gjør

```
a[index] = 4 + 2
```

Vanlig invariant: Pilen peker på første tegn etter det siste token som ble lest.
Ved + skal da denne pekeren flyttes!



Tegnet pilen peker på ligger gjerne i en gitt variabel 'curChar' e-lign.



INF 5110 - 2012

Fortran

Spesielle regler for oppdeling i tokens

```
I F ( X 2 . EQ. 0) THE N
```

```
IF(X2.EQ.0)THEN
```

```
IF(IF.EQ.0)THENTHEN=1.0
```

```
DO99I=1,10
```

```
DO 99 I=1,10
 -
 -
 -
99 CONTINUE
```

```
DO99I=1.10
```

INF 5110 - 2012

Klassifisering

- Hva som er en god klassifisering blir ikke klart før senere i kompilatoren
- En mulig regel: Det som skal behandles likt under syntaktisk analyse skal i samme klasse.
- Selve tokenet angir klassen
  - hvilket leksem det er angis ved attributter
- Det enkleste er å angi selve teksten (f.eks. string value) som attributt
- Men ofte vil scanneren også:
  - Sette navn i tabell, og gi med indeksen som attributt
  - Sette tekst-konstanter i tabell, og angi indeksen som attributt
  - beregne tallkonstanter, og angi verdien som attributt
  - …

En mulig klassifisering

- Navn (identifikator): abc25
- Heltallskonstant: 1234
- Reell konstant: 3.14E3
- Boolsk konstant: true false
- Tekst-konstant: ”dette er en tekstkonstant”
- Aritmetisk operator: + - * /
- Relasjons-operator: < <= >= > = <>
- Logisk operator: and or not
- Alle andre tokens i hver sin gruppe, f.eks.:
  - nøkkelord ( ) [ ] { } := , ; .

- MEN: Langt fra klart at denne blir den beste ……

C-typer som kan representere et token

```
typedef struct
    { TokenType tokenval;
      char * stringval;
      int numval;
    } TokenRecord;
```

Hovedklassifikasjonen

Selve teksten,
eller verdien

Hvis man vil lagre bare ett attributt:

```
typedef struct
    { TokenType tokenval;
      union
      { char * stringval;
        int numval;
      } attribute;
    } TokenRecord;
```

En scanner er ikke stor og vanskelig

- De forskjellige token-klassene kan lett beskrives i prosa-tekst
- Ut fra det kan en scanner skrives rett fram, uten særlig teori
- Kan typisk ta noen hundre linjer, der det samme prinsippet stadig går igjen
- Men, man kan ønske seg:
  - Å angi token-klassene i en passelig formalisme
  - Ut fra denne å automatisk få laget et scanner-program
- Det er dette kapittel 2 handler mest om

Framgangsmåte

for automatisk å lage en scanner

- Beskriv de forskjellige token-klassene som regulære uttrykk
  - Eller litt mer fleksibelt, som regulære definisjoner
- Omarbeid dette til en NFA (Nondeterministic Finite Automaton)
  - Er veldig rett fram
- Omarbeid dette til en DFA (Deterministic Finite Automaton)
  - Dette kan gjøres med en kjent, grei algoritme
- En DFA kan uten videre gjøres om til et program
- Det er hele veien et kompliserende element at vi:
  - ikke bare skal finne ett token, men en sekvens av token
  - hvert token skal være så langt som mulig
- Vi skal se på dette i følgende rekkefølge:

  (1) Regulære uttrykk (2) DFAer (3) NFAer

Definisjon av regulære uttrykk

- Og det språket de definerer = mengden av strenger

A **regular expression** is one of the following:

1. A **basic** regular expression, consisting of a single character **a**, where a is from an alphabet Σ of legal characters; the metacharacter $\boldsymbol{\varepsilon}$; or the metacharacter $\boldsymbol{\phi}$. In the first case, $L(\mathbf{a}) = \{a\}$; in the second, $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$; in the third, $L(\boldsymbol{\phi}) = \{\,\}$.
2. An expression of the form $\boldsymbol{r} \mid \boldsymbol{s}$, where r and s are regular expressions. In this case, $L(\boldsymbol{r} \mid \boldsymbol{s}) = L(r) \cup L(s)$.
3. An expression of the form rs, where r and s are regular expressions. In this case, $L(rs) = L(r)L(s)$.
4. An expression of the form r*, where r is a regular expression. In this case, $L(r^*) = L(r)^*$.
5. An expression of the form (r), where r is a regular expression. In this case, $L((r)) = L(r)$. Thus, parentheses do not change the language. They are used only to adjust the precedence of the operations.

Presedens:
*, konkatenering,|

$L(\varepsilon)$ = språket som bare inneholder ε-strenger
$L(\varnothing)$ = språket uten noen strenger

Eksempler - I

$\Sigma=\{a,b,c\}$

- Strenger som har nøyaktig en b

  $(a \mid c)^* \; b \; (a \mid c)^*$

- Strenger som har maks en b

  $(a \mid c)^* \mid (a \mid c)^* \; b \; (a \mid c)^*$

  $(a \mid c)^* \; (b \mid \varepsilon) \; (a \mid c)^*$

- Strenger som har formen aaaabaaaa (dvs like mange a-er)

  $(a^n \; b \; a^n)$?

Eksempler - II

- Strenger som ikke inneholder to b-er etter hverandre

(b (a | c))*	en a eller c etter hver b
((a | c)* | (b (a | c))*)*	kombinert med (a | c)*
((a | c) | (b (a | c)))*	forenklet
(a | c | ba | bc)*	enda mer forenklet
(a | c | ba | bc)* (b | ε)	får med b på slutten
(*notb* | b *notb*)* (b | ε)	hvor *notb* = a | c

INF 5110 - 2012

Mer rasjonelle skrivemåter

- r+ = rr*
- r? = r | ε

- Gi regulære uttrykk navn (regulære definisjoner), og så bruke disse navnene videre i regulære uttrykk:
  - digit = [0-9]
  - nat = digit+
  - signedNat = (+|-)nat
  - number = signedNat (”.” nat)?(E signedNat)?

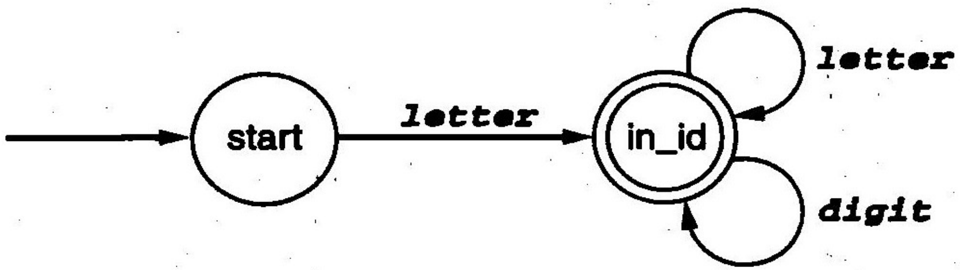
- Spesielle skrivemåter for tegnmengder
  - [0-9] [a-z]
  - ~a ikke a ~(a | b) hverken a eller b
  - . hele Σ .* en vilkårlig streng

Deterministisk Endelig Automat

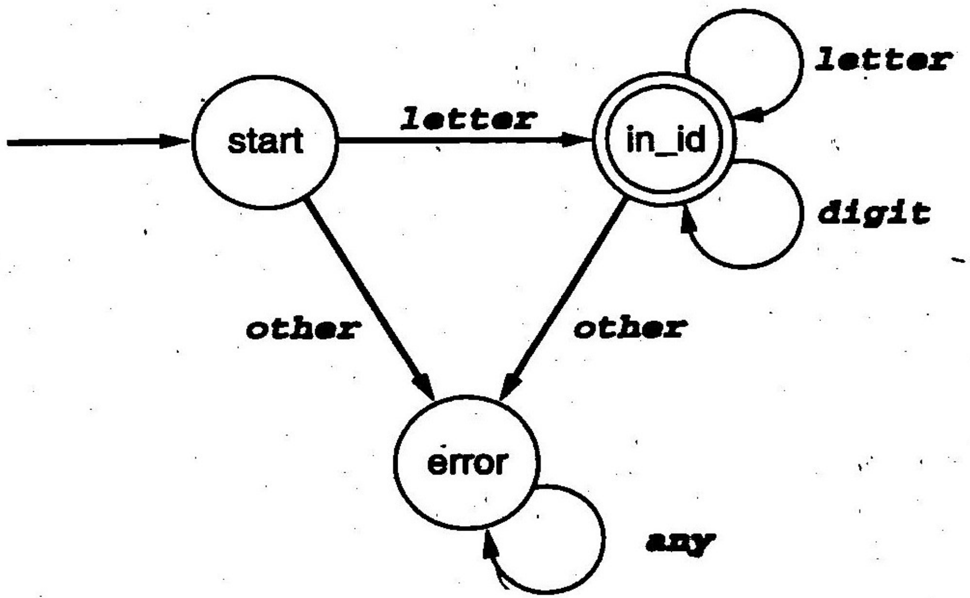
A **DFA** (deterministic finite automaton) M consists of an alphabet Σ, a set of states S, a transition function T: $S \times \Sigma \rightarrow S$, a start state $s_0 \in S$, and a set of accepting states $A \subset S$. The language accepted by M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2. . .c_n$ with each $c_i \in \Sigma$ such that there exist states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$ with s_n an element of A (i.e., an accepting state).

DFA

identifier = *letter* (*letter* | *digit*)*



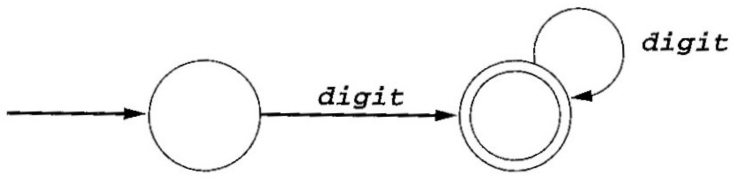
Funksjonen T: S×Σ -> S er ikke fullstendig definert

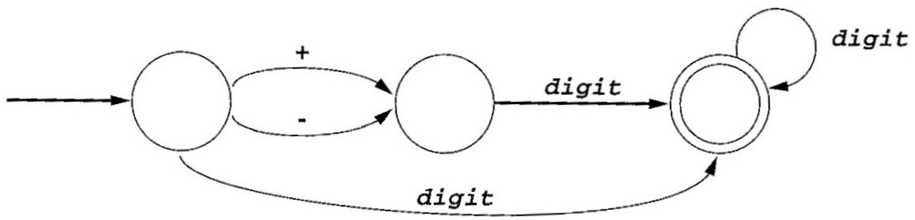


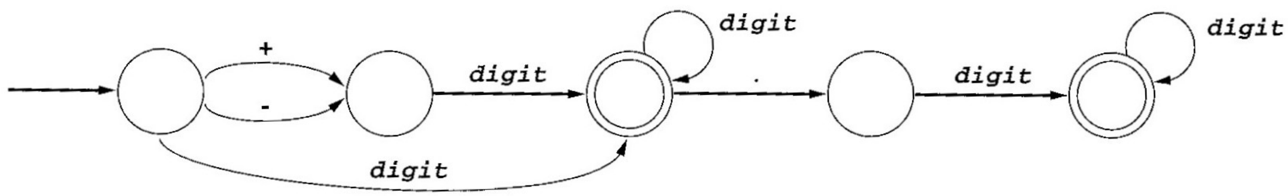
Denne utvidelsen (med en feiltilstand) er underforstått

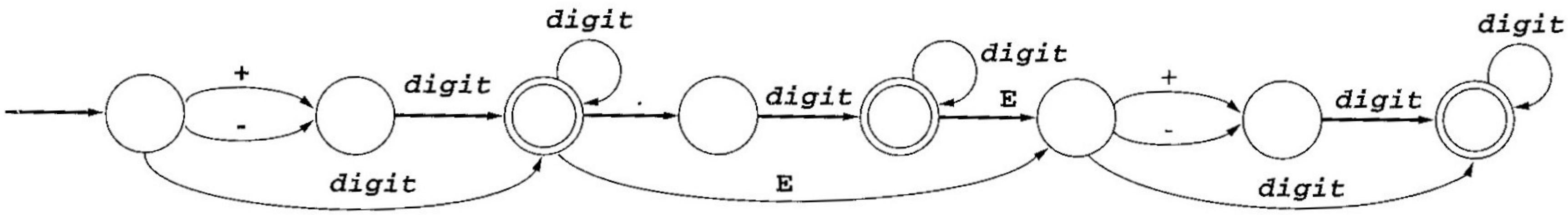
DFA for tall

Regulære definisjoner
digit = [0-9]
nat = digit+
signedNat = (+|-)? nat
number =
signedNat (“.”nat)?(E signedNat)?



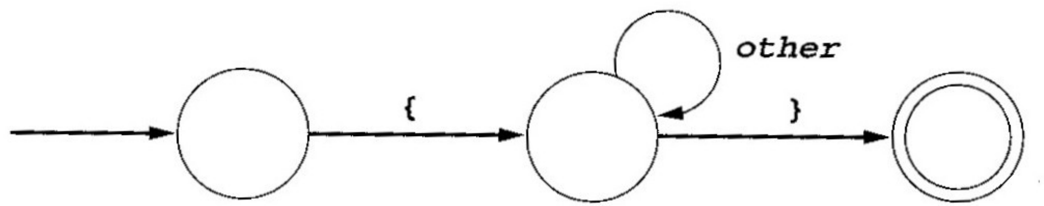




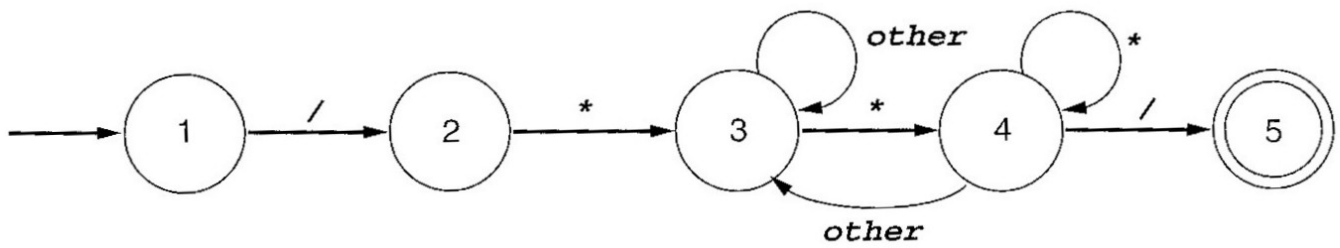


DFA for kommentarer

Pascal type



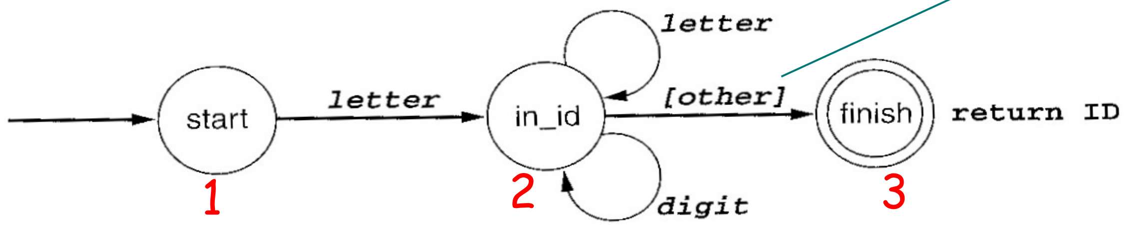
C, C++, Java type



INF 5110 - 2012

Implementasjon 1 av DFA

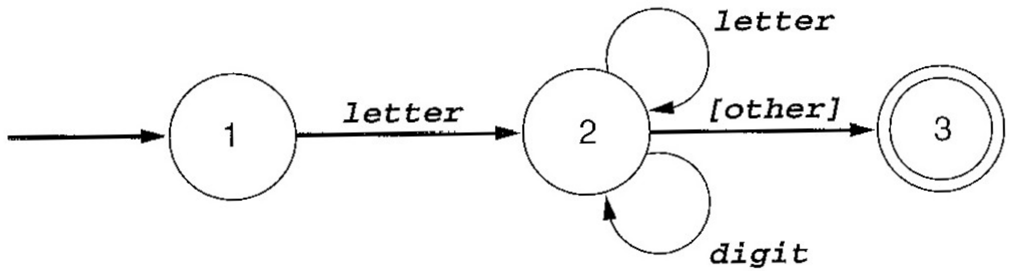
ikke flytt til neste tegn



{ starting in state 1 }
if *the next character is a letter* **then**
  advance the input;
  { now in state 2 }
  while *the next character is a letter or a digit* **do**
    advance the input; { stay in state 2 }
  end while;
  { go to state 3 without advancing the input }
  accept;
else
  { error or other cases }
end if;

er neste tegn et siffer: tall
er neste tegn {+,-,*,/}: arit. operator
...
...

Implementasjon 2 av DFA



Tilstanden eksplisitt representert ved et tall

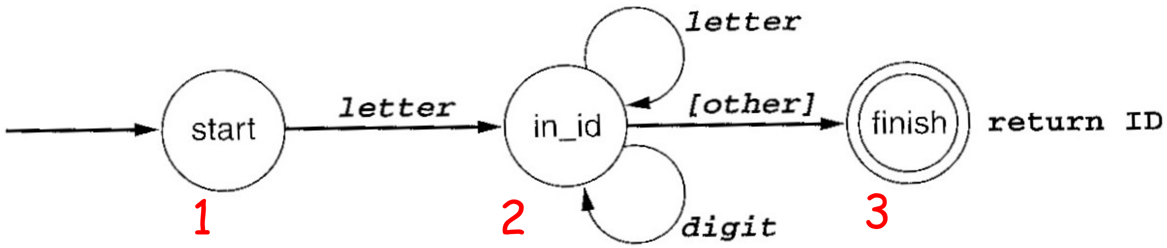
```
state := 1; { start }
while state = 1 or 2 do
  case state of
  1:   case input character of
       letter : advance the input;
                state := 2;
       else state := . . . { error or other };
       end case;
  2:   case input character of
       letter, digit: advance the input;
                      state := 2; { actually unnecessary }
       else state := 3;
       end case;
  end case;
end while;
if state = 3 then accept else error ;
```

hvis det bare er navn vi leter etter

hvis vi også aksepterer tall, vil siffer føre oss til en ny lovlig tilstand

Implementasjon 3 av DFA

- Har et fast program
- Automaten ligger i en tabell



state \ input char	letter	digit	other
1	2		
2	2	2	3
3			

state := 1;
ch := *next input character;*
while not *Accept*[state] **and not** *error(state)* **do**
  newstate := *T*[*state*,*ch*];
  if *Advance*[*state*,*ch*] **then** *ch* := *next input char;*
  state := *newstate;*
end while;
if *Accept*[*state*] **then** *accept;*

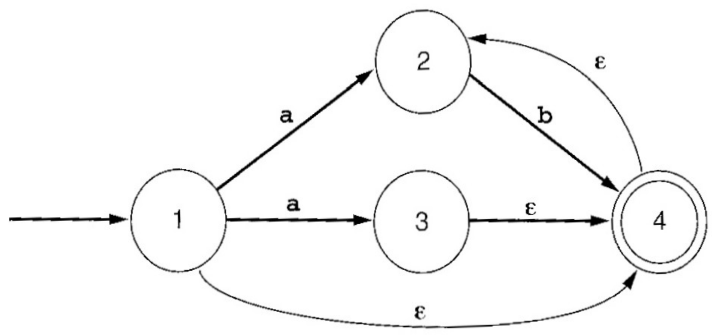
state \ input char	letter	digit	other	*Accepting*
1	2			no
2	2	2	[3]	no
3				yes

Definisjon av NFA

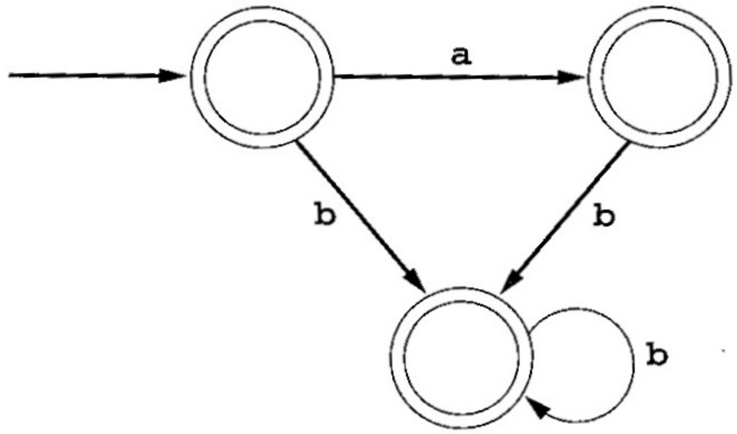
An **NFA** (nondeterministic finite automaton) M consists of an alphabet Σ, a set of states S, a transition function $T: S \times (\Sigma \cup \{\varepsilon\}) \to \wp(S)$, a start state s_0 from S, and a set of accepting states A from S. The language accepted by M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2. . .c_n$ with each c_i from $\Sigma \cup \{\varepsilon\}$ such that there exist states s_1 in $T(s_0, c_1)$, s_2 in $T(s_1, c_2)$, . . . , s_n in $T(s_{n-1}, c_n)$ with s_n an element of A.

- Kan ofte være lett å sette opp, spesielt ut fra et regulært uttrykk
- Kan ses på som syntaks-diagrammer

NFA



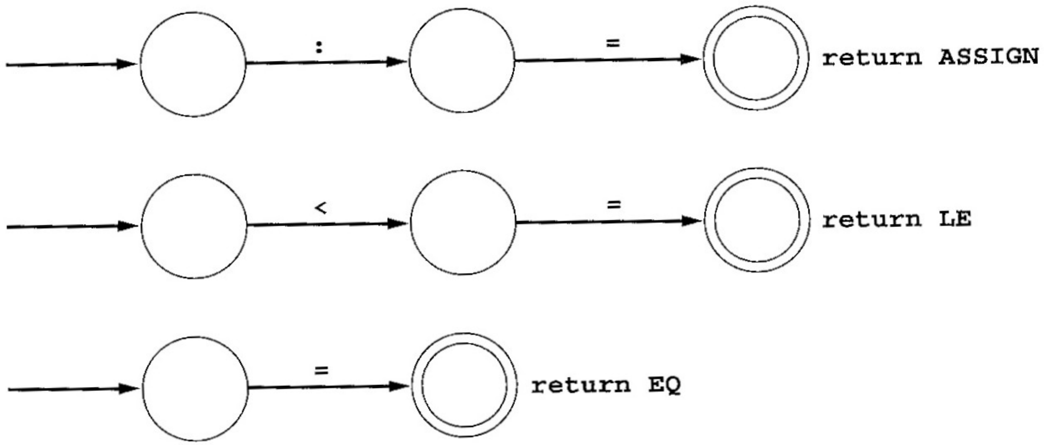
DFA



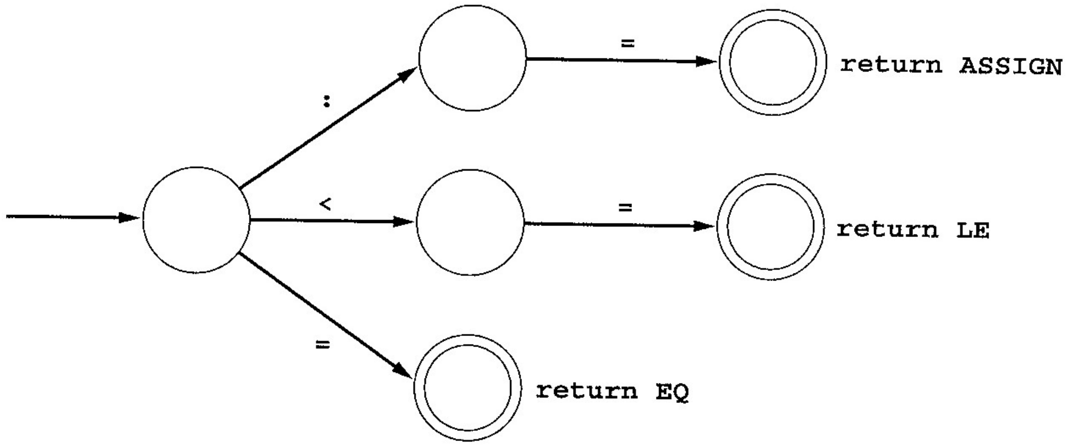
- ikke greit å gjøre til algoritme
- beskriver samme språk

INF 5110 - 2012

Motivasjon for å innføre NFA-er (1)

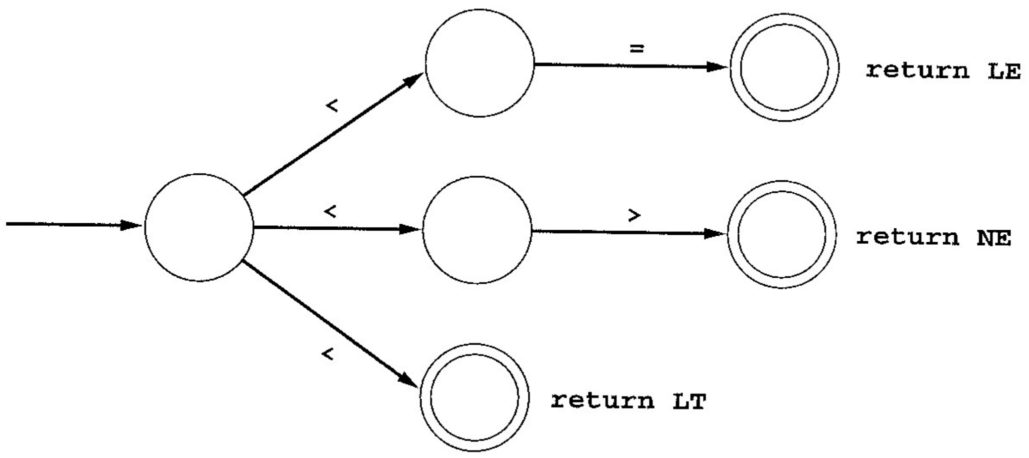


DFA-er
for de
enkelte
token

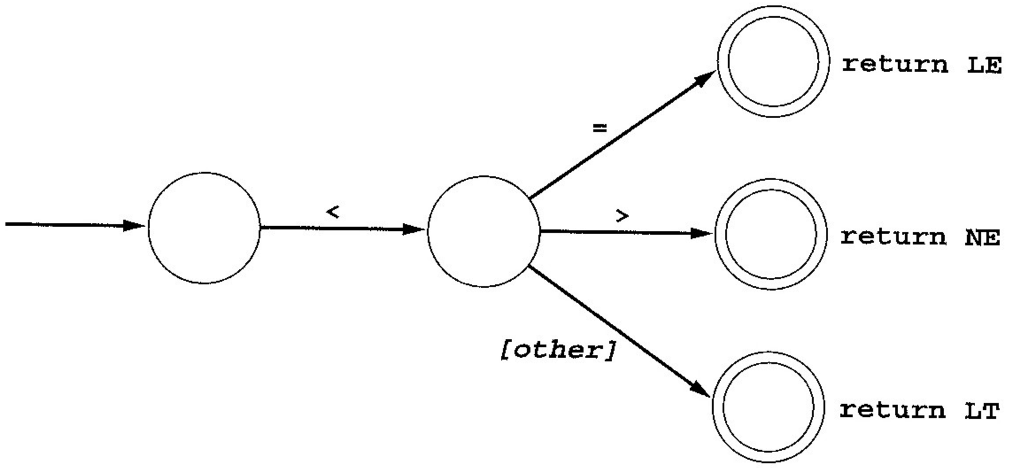


Siden de
starter
forskjellig går
det greit å slå
dem sammen,
men de
opprindelige
automater er
der ikke lenger

Motivasjon for å innføre NFA-er (2)

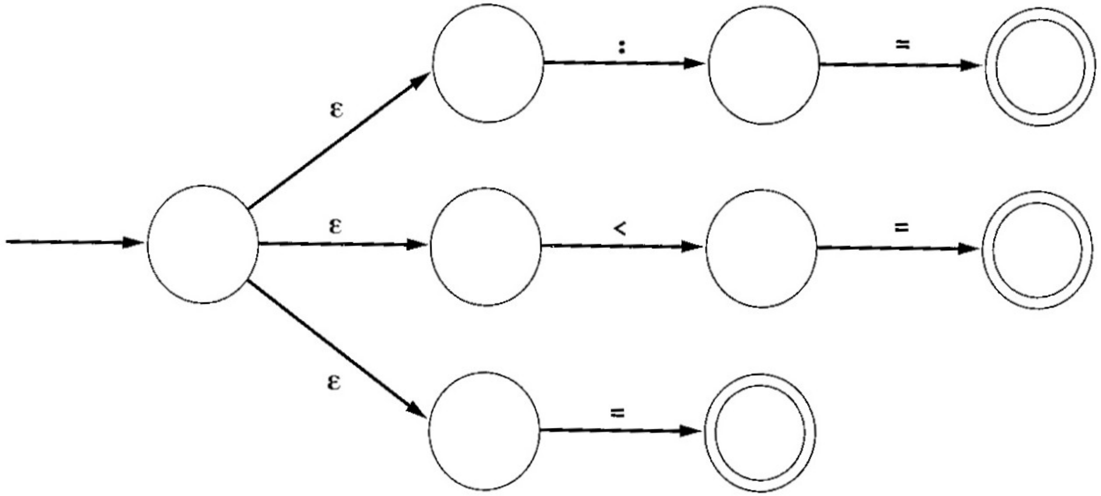


Med disse token-definisjoner går det ikke med en enkel sammenslåing, da dette ikke er en DFA



I dette tilfellet går det an å slå sammen på første tegn, men dette er ikke altid mulig

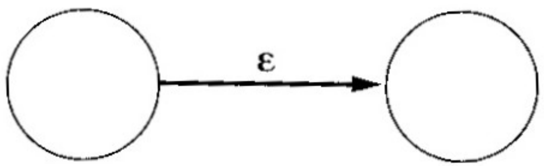
Motivasjon for å innføre NFA-er (3)



Ville
være
greiere
generelt
å gjøre
det slik

Derfor

(1) Innfører ε-kant



(2) Fjerner kravet om bare én a-kant ut fra hver tilstand

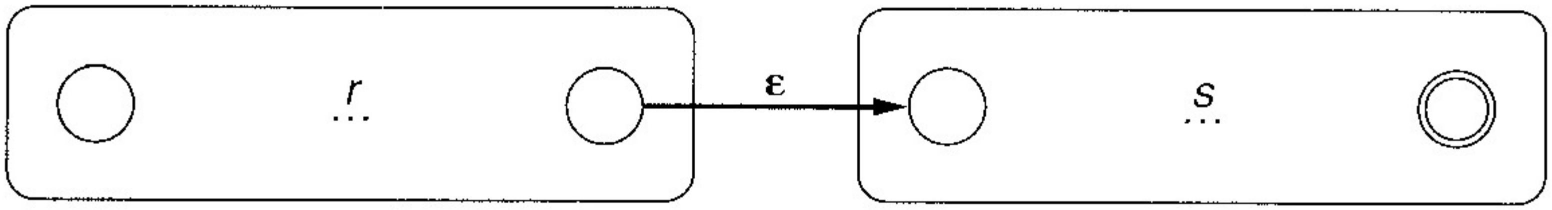
Thomson-konstruksjon I

(Ethvert regulært uttrykk skal bli automat på denne formen:

a:

ε:

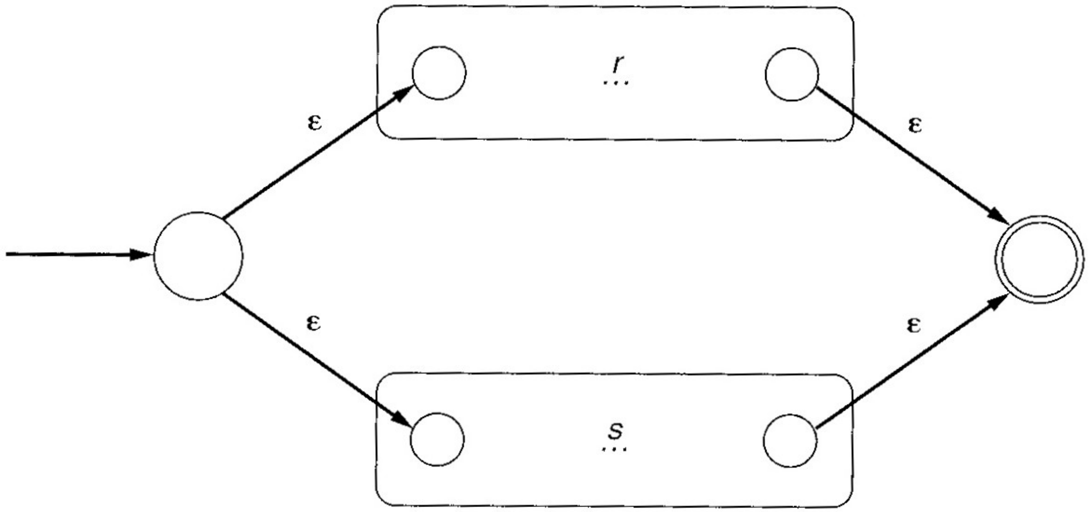
rs:



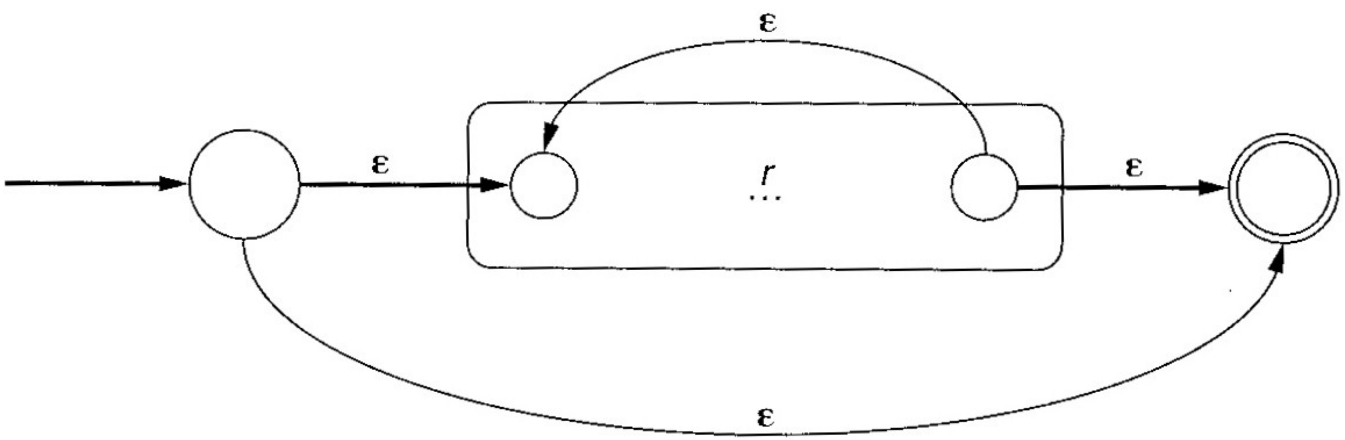
INF 5110 - 2012

Thomson-konstruksjon II

r | s:



r*:



Eksempel Thomson-konstruksjon

ab | a