# Dagens tema

- Syntaks
  (Komp. 47, kap. 3 (og noe 4))
  - Repetisjon
  - Regulære språk i klassisk BNF
  - Regulære språk i utvidet BNF
  - Regulære språk i jerbanediagrammer
  - Regulære språk og automater
  - Fra ikke-deterministisk automat til deterministisk

# Repetisjon

Sist så vi ulike notasjoner for syntaks:

- Jernbanediagrammer
- BNF-grammatikker
- Utvidet BNF
- Endelige, deterministiske automater
- Endelige, ikke-deterministiske automater

Vi så også hvordan slike ble brukt til å *generere* eller *godkjenne* strenger fra språket.

Inndeling av språk som kan beskrives med BNF-grammatikk:

- **Type 3**-språk («regulære språk») har ett metasymbol på venstresiden og kun grunnsymboler på høyresiden, eventuelt med et metasymbol til sist.

$$\langle \textit{binærtall} \rangle \rightarrow \mathbf{0} \mid \mathbf{0}\ \langle \textit{binærtall} \rangle \mid \mathbf{1} \mid \mathbf{1}\ \langle \textit{binærtall} \rangle$$

  Slike språk brukes til å angi søkekriterier (regulære uttrykk) i Perl, Emacs, grep, egrep, awk, m.fl.

### Hvorfor brukes regulære språk til søking?

1. Det er enkelt å angi et ganske kraftig søkekriterium.
2. Det er lett å lage en meget rask **automat** som sjekker lovlige uttrykk.

- **Type 2**-språk («kontekst-frie») har bare ett metasymbol på venstresiden.

  Omtrent alle programmeringsspråk benytter en kontekst-fri grammatikk til å definere språkets syntaks.

  — Det gir en klar men lettlest definisjon av syntaksen.

  — Det er en basis for å skrive en syntakstolker («parser»).

- **Type 1**-språk («kontekst-sensitive») krever at høyresiden er minst like lang som venstresiden.

  Dette gjør det mulig å sjekke navnebindinger og finne typefeil. Ble brukt til Algol-68 men lite siden.

- **Type 0**-språk har ingen restriksjoner.

  Disse har bare teoretisk interesse.

## Ulike representasjoner av regulære språk

La oss som eksempel se på et regulært språk for binære tall med **binærer**. Lovlige ord er

0 1 101 0.10 100.1010 10.1

Imidlertid er det ikke lov med ledende 0-er eller binærpunktum uten foregående eller etterfølgende sifre, så følgende er ikke tillatt:

001 10. .01

# Ulike representasjoner av regulære språk

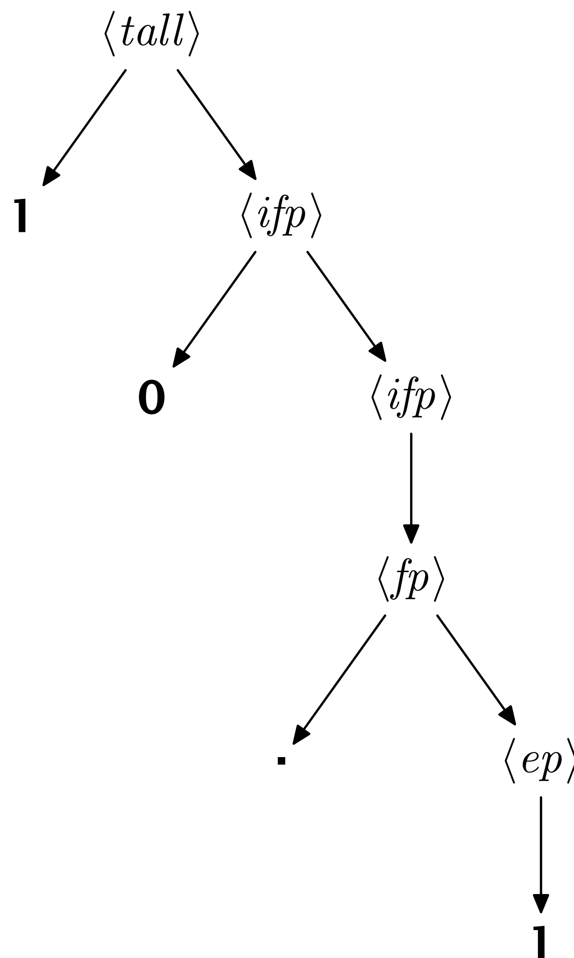
## Representasjon 1: Klassisk BNF

$$
\begin{aligned}
\langle tall \rangle &\rightarrow \mathbf{0}\ \langle fp \rangle \mid \mathbf{1}\ \langle ifp \rangle \\
\langle ifp \rangle &\rightarrow \mathbf{1}\ \langle ifp \rangle \mid \mathbf{0}\ \langle ifp \rangle \mid \langle fp \rangle \\
\langle fp \rangle &\rightarrow \varepsilon \mid \mathbf{.}\ \langle ep \rangle \\
\langle ep \rangle &\rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{0}\ \langle ep \rangle \mid \mathbf{1}\ \langle ep \rangle
\end{aligned}
$$

(Symbolet «$\varepsilon$» betegner et tomt alternativ.)

## Parseringstrær

Vi sjekker om et uttrykk er lovlig i følge grammatikken ved å se om det lar seg gjøre å lage et **parseringstre**:



## Representasjon 2: Utvidet BNF

I «utvidet BNF» krever vi for et regulært språk at det *ikke* er metasymboler i høyresiden. Denne høyresiden kalles da et **regulært uttrykk**.

Definisjonen av $\langle tall \rangle$ som regulært uttrykk blir

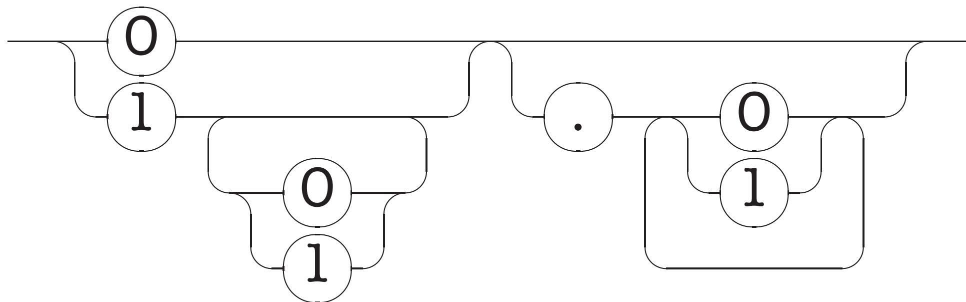
$$\langle tall \rangle \rightarrow [\![ \mathbf{0} \mid \mathbf{1} \; [\![ \mathbf{0} \mid \mathbf{1} ]\!]^{*} ]\!] \; [\![ \mathbf{.} \; [\![ \mathbf{0} \mid \mathbf{1} ]\!]^{+} ]\!]^{?}$$

**Notasjon**
Selv om essensen er den samme, kan notasjonen for regulære uttrykk variere fra ett program til et annet.

# Representasjon 3: Jernbanediagram

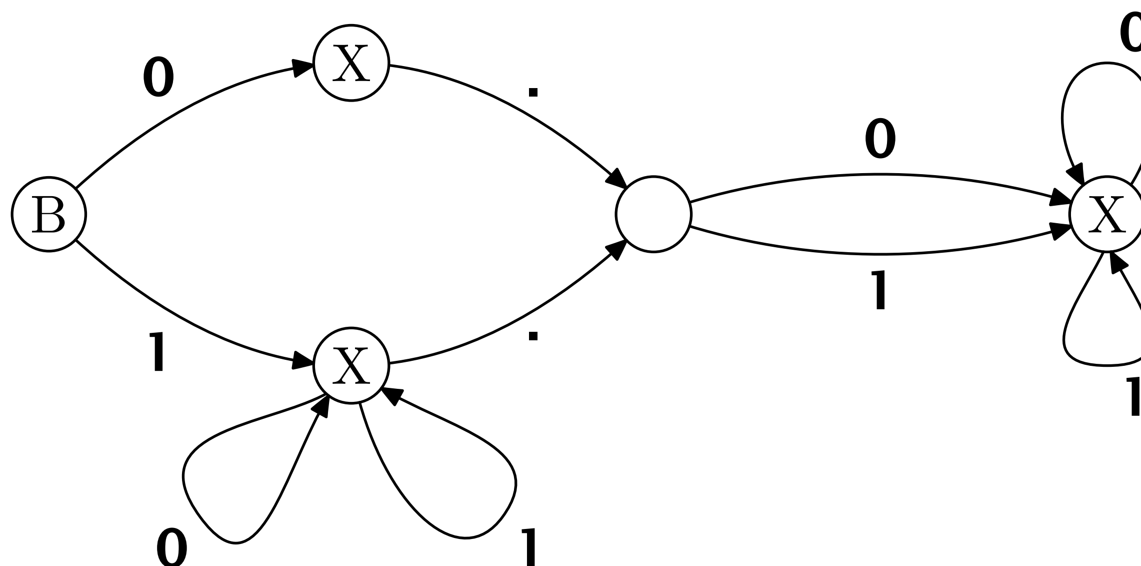
Siden jernbanediagram bare er en mer visuell form for utvidet BNF, kan vi angi en regulær grammatikk med et diagram hvor det ikke finnes metasymboler.

*tall*



# Representasjon 4: Deterministisk automat

Her definerer vi en **deterministisk automat**) hvor tegnene fører oss fra én **tilstand** til neste.




Tilstanden merket «B» er starttilstanden, men de med «X» er lovlig slutttilstander.

## Hvorfor er deterministisk automater så nyttige?

En deterministisk automat kan lett representeres av en matrise som angir neste tilstand.

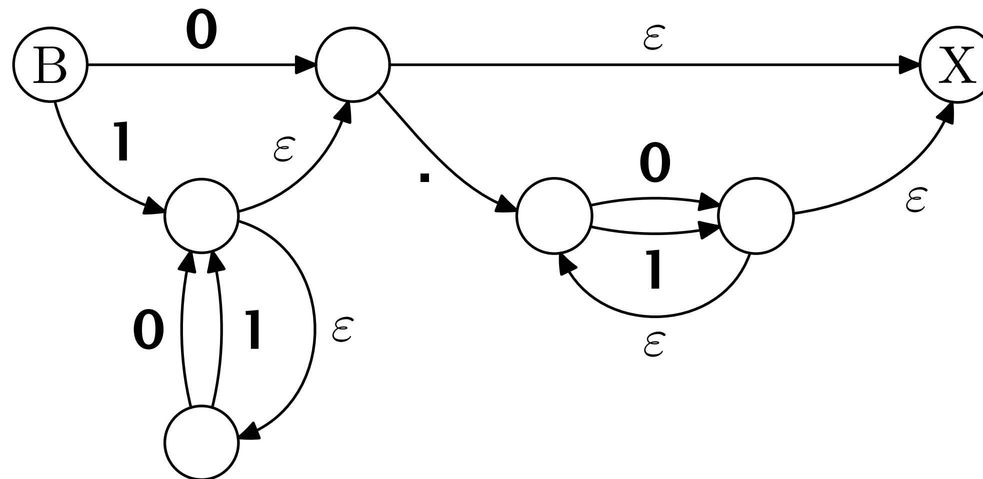
| Tilstand | 0 | 1 | . | Slutt |
|---|---|---|---|---|
| 1 | 2 | 3 | F | |
| 2 | F | F | 4 | Ja |
| 3 | 3 | 3 | 4 | Ja |
| 4 | 5 | 5 | F | |
| 5 | 5 | 5 | F | Ja |
| F | F | F | F | |

Det er vanlig å innføre en ekstra tilstand «F» for feil.

# Hvordan lage en deterministisk automat?

Det finnes en enkelt algoritme for å lage en deterministisk automat.

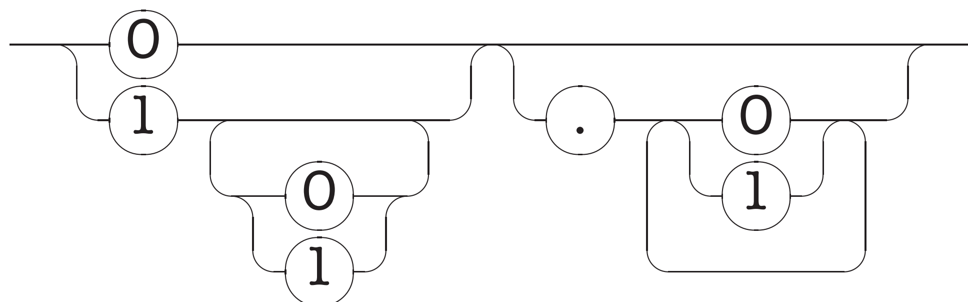
Først lages en ikke-deterministisk automat:

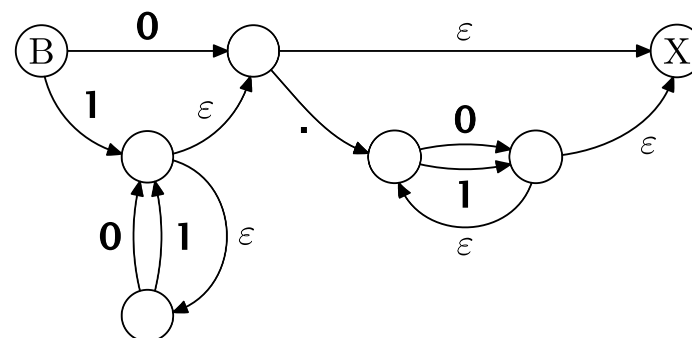


# Hvordan lage en ikke-deterministisk automat?

Ta utgangspunkt i jernbanediagrammet:

*tall*




1. Hver «pens» blir til en node i den ikke-deterministiske automaten.
2. Hvert sluttsymbol blir en merket kant. Noen får et tomt symbol ($\varepsilon$).
3. Merk nodene hvor man skal begynne og slutte.

**Hvordan bruke en ikke-deterministisk automat?**
En ikke-deterministisk automat er dårlig egnet til å sjekke tekst.

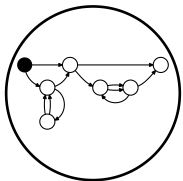
- Det kan gå flere kanter med samme merke fra en node.
- Det er vanskelig å gjette når man skal følge en kant med tomt symbol ($\varepsilon$).

Imidlertid kan man lage en **deterministisk automat** utifra en ikke-deterministisk.

# Hvordan lage en deterministiske automater

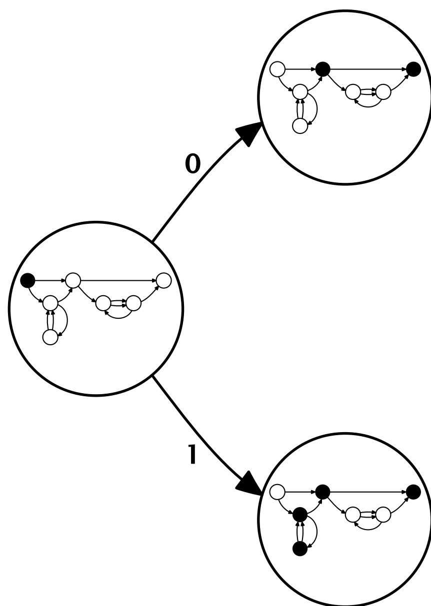
Start med startnodene. (Dette er startnoden samt de man kan komme til langs kanter med tomt symbol.)

Lag en tilstand for disse nodene.

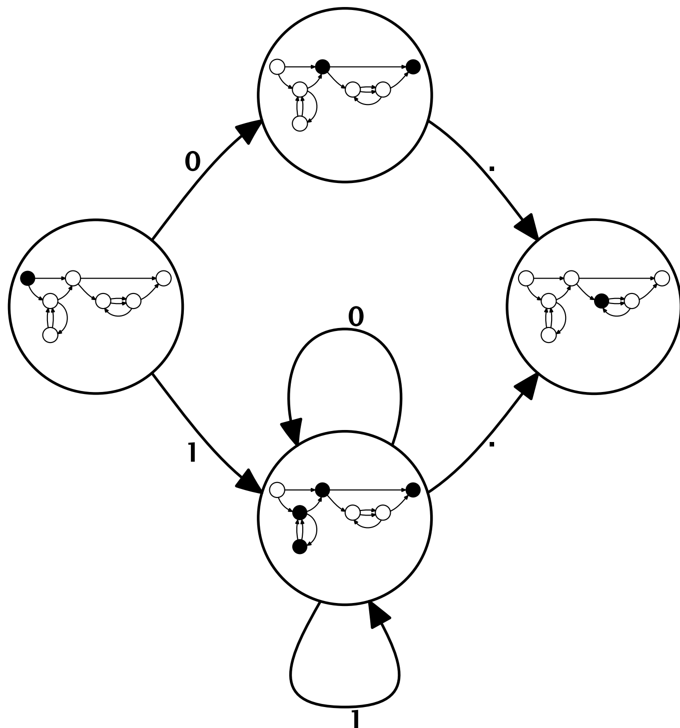


(Teknikken er basert på å lage tilstander som representerer et utvalg av noder i den ikke-deterministiske automaten. De utvalgte nodene er sorte.)

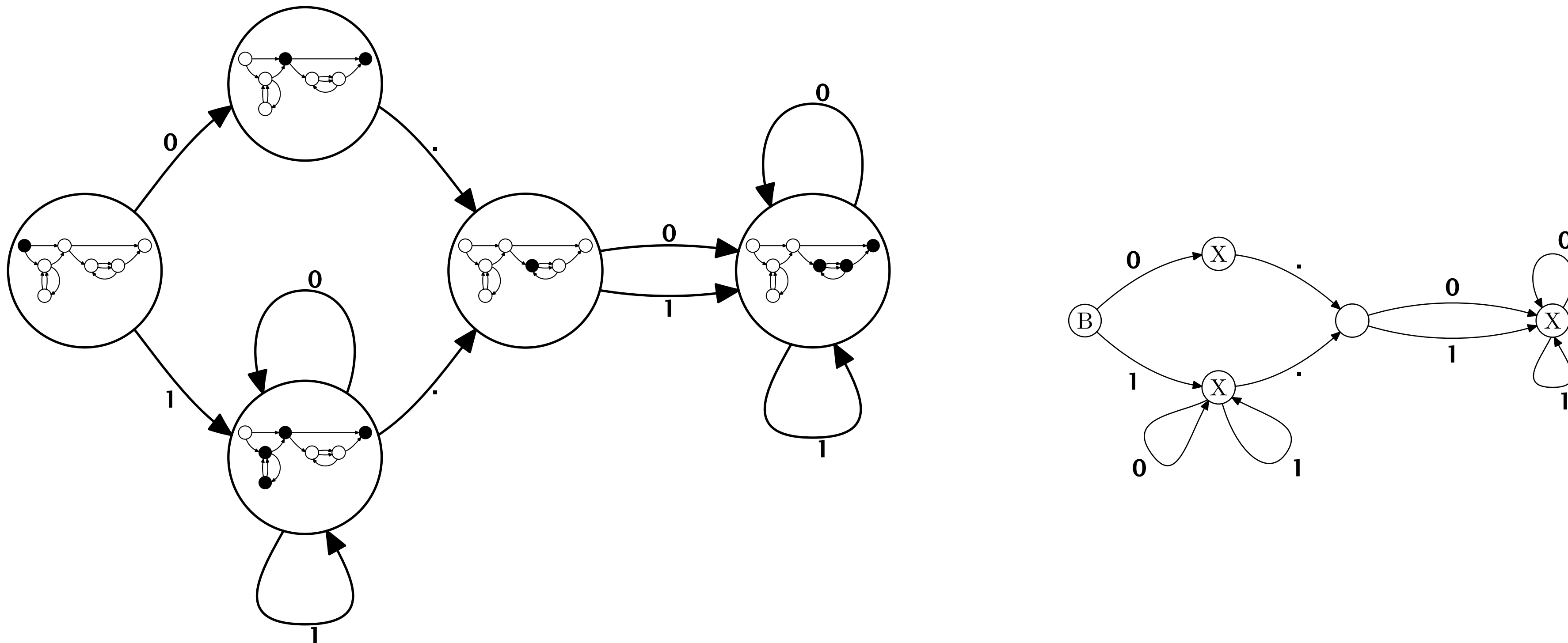
Fra hver tilstand følg kantene med ulike terminalsymboler (og tomme kanter). Utvide den deterministiske automaten med disse.



Fortsett med dette.



Til slutt har man den ferdige deterministiske automaten.



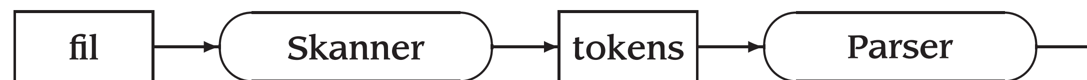

Vi ser at den er ekvivalent med den vi hadde.

# Syntaksanalyse

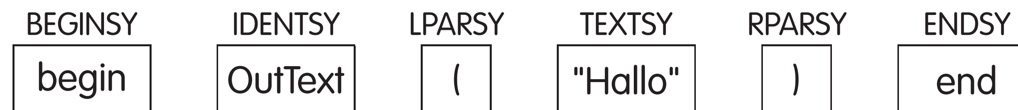
En *parser* er et program som analyserer en tekst i henhold til en syntaks. Det sjekker om data er lovlig, og identifiserer de ulike elementene. Alle kompilatorer og interpreter inneholder en parser.

## Skanner

Det er vanlig at en parser har en «preprosessor» som kalles en *skanner*:




Den setter sammen tegn til symboler (ofte kalt *tokens*), for eksmpel



# Parsering

Parsering vil si å sjekke at en setning (eller et program) er syntaktisk riktig, det vil si å konstruere det tilhørende syntakstreet.
(I praksis bryr vi oss gjerne ikke om å *faktisk* konstruere syntakstreet, vi nøyer oss med å vite at vi kunne laget et hvis vi hadde hatt behov for det.)

Generelt ønsker vi å kunne konstruere treet ved å lese setningen *en* gang, fra venstre mot høyre.

Eksempelgrammatikk:

$$\langle \mathit{uttrykk} \rangle \rightarrow \langle \mathit{uttrykk} \rangle \; \mathbf{+} \; \langle \mathit{term} \rangle \mid \langle \mathit{term} \rangle$$

$$\langle \mathit{term} \rangle \rightarrow \langle \mathit{term} \rangle * \mathbf{navn} \mid \mathbf{navn}$$

Ut fra denne grammatikken skal vi se på parsering av setningen:

navn * navn + navn

# Top-down parsering

Her konstrueres treet ovenfra og ned, det vil si at vi starter med startsymbolet i roten, og så forsøker å avlede den aktuelle setningen ut fra dette:

UTTRYKK

navn * navn + navn

# Bottom-up parsering

Her konstruerer vi treet nedenfra og opp. Vi starter da med å finne noe i setningen som tilsvarer høyresiden i en produksjon, og *reduserer* denne delsetningen til det tilhørende metasymbolet. Målet er da å redusere seg tilbake til startsymbolet:

UTTRYKK

navn * navn + navn

# LL(1)-parsering

LL(1)-parsering er en top-down strategi der vi foretar en *venstreavledning* fra startsymbolet.

## Recursive descent

- Til hvert metasymbol svarer en metode.
- Metoden tar seg av det ene metasymbolet, men kan kalle andre metoder.
- Når metoden kalles, skal skanneren inneholde første symbol i den aktuelle produksjonen (for syntaktisk korrekt setning).
- Når metoden er ferdig, skal skanneren inneholde første symbol etter den leste teksten.