Dagens tema

Grundig repetisjon og utdyping:

- Syntaks kontra semantikk
- Regulære uttrykk og automataer
- Ulike typer språk
- Ulike representasjoner av regulære språk
- Endelige tilstandsmaskiner (FSM-er)
  - Deterministiske FSM-er
  - Ikke-deterministiske FSM-er

Hva er syntaks?

En overskrift i en norsk avis:

Fanger krabber så lenge de orker

Er det i C lov å skrive

for (;;) { . . . } ✔
while () { . . . } ✘

for å få en evig løkke?

I C, betyr *p++ (*p)++ ✘
eller *(p++)? ✔

Hvilke av disse er lovlige flyt-tall i Java:

3	✘
3.14	✔
6.28e−18	✔
.1	✔
22.D	✔
e3	✘
1.2e+2.0	✘

Syntaksen (dvs **grammatikken**) gir svaret.

Semantikk

Syntaks forteller ikke alt om et språk:

Per gikk bort til Anne og løftet ham opp.

Tilsvarende har vi i programmeringsspråk:

```
integer a, b;
boolean c;

a := b + c;
```

Semantikken forteller oss

- Hvordan skjer navnebinding?
- Hva er «meningen» med et program (mao hva er resultatet av å kjøre det)?

Dessverre finnes intet enkelt språk (à la BNF) for å beskrive et språks semantikk, men les om *Denotational semantics* i avsnitt 4.3 i læreboken.

I et veldesignet språk er det godt samsvar mellom semantikken og brukerens intuitive oppfatning. Ingen spåk er imidlertid perfekte. På nynorsk har vi

Jentungen møtte løva, og ho åt han.

I C har vi at

c = 2; f(c, c++);

gir enten f(2,2); eller f(3,2);.

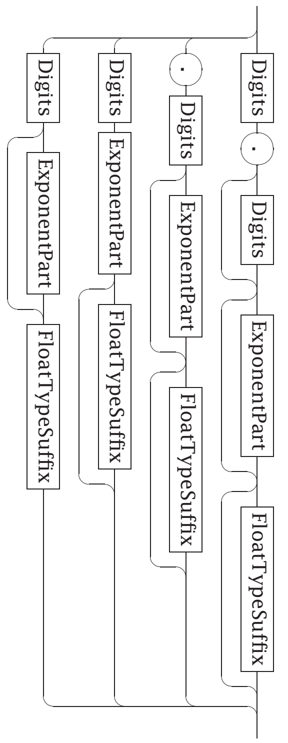
I Java har vi

```
class ClassA {
   int n;

   void print (ClassA x) {
      System.out.println("n="+n+" og x.n="+x.n);
   }

   public static void main (String arg[]) {
      ClassA a = new ClassB(), b = new ClassB();
      a.n = 1;  b.n = 2;
      a.print(b);
   }
}

class ClassB extends ClassA {
   int n2;

   void print (ClassB x) {
      System.out.println(n==x.n ? "Like" : "Ulike");
   }
}
```

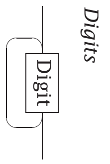
Javas definisjon av flyt-tall

Dette er fra *The Java Language Specification*:

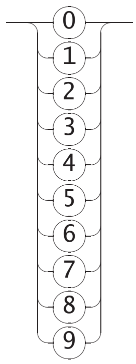
FloatingPointLiteral



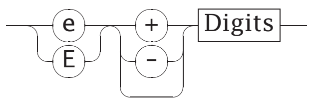
Digits



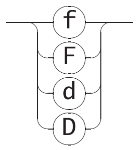
Digit



ExponentPart



FloatTypeSuffix



I slike **jernbanediagrammer** har vi:

Terminalsymboler (også kalt **grunnsymboler**) i runde bokser er symboler som forekommer i brukerens tekst.

Metasymboler i firkantede bokser blir definert et annet sted.

Produksjoner er definisjoner av metasymboler.

Takket være metasymbolene kan vi

- forenkle diagrammet,
- spare kode ved gjentagelse, og
- lage rekursive definisjoner.

Jernbanediagrammer er lette å lese, men de har noen ulemper:

- De tar stor plass.
- De kan ikke uttrykkes som vanlig tekst.
- De må enten håndtegnes eller lages av spesiell programvare.†

† Mine diagrammer er laget med LaTeX-pakken rail.

BNF

«Backus Normal Form» (eller «Backus-Naur form») ble laget til Algol-60 og er den vanligste notasjonen for syntaks.

$$
\begin{array}{lcl}
\langle\text{FloatingPointLiteral}\rangle & \rightarrow & \langle\text{Digits}\rangle\ \texttt{.}\ \langle\text{Digits}\rangle^{?}\ \langle\text{ExponentPart}\rangle^{?}\ \langle\text{FloatTypeSuffix}\rangle^{?} \\
\langle\text{FloatingPointLiteral}\rangle & \rightarrow & \texttt{.}\ \langle\text{Digits}\rangle\ \langle\text{ExponentPart}\rangle^{?}\ \langle\text{FloatTypeSuffix}\rangle^{?} \\
\langle\text{FloatingPointLiteral}\rangle & \rightarrow & \langle\text{Digits}\rangle\ \langle\text{ExponentPart}\rangle\ \langle\text{FloatTypeSuffix}\rangle^{?} \\
\langle\text{FloatingPointLiteral}\rangle & \rightarrow & \langle\text{Digits}\rangle\ \langle\text{ExponentPart}\rangle^{?}\ \langle\text{FloatTypeSuffix}\rangle \\
\langle\text{Digits}\rangle & \rightarrow & \langle\text{Digit}\rangle^{+} \\
\langle\text{Digit}\rangle & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\langle\text{ExponentPart}\rangle & \rightarrow & [\![\ \texttt{e} \mid \texttt{E}\ ]\!]\ [\![\ \texttt{+} \mid \texttt{-}\ ]\!]^{?}\ \langle\text{Digits}\rangle \\
\langle\text{FloatTypeSuffix}\rangle & \rightarrow & \texttt{f} \mid \texttt{F} \mid \texttt{d} \mid \texttt{D}
\end{array}
$$

INF3110/4110

Følgende gjelder for denne formen for BNF:

- Metasymboler skrives slik: ⟨Digit⟩.
- Terminalsymboler skrives slik: `begin`.
- Det kan være flere definisjoner for hvert metasymbol.
- Følgende spesialsymboler brukes:

  | skiller alternativer.

  $^{?}$ angir at noe kan forekomme 0 eller 1 gang.

  * angir at noe kan forekomme 0 eller flere ganger.

  $^{+}$ angir at noe kan forekomme 1 eller flere ganger.
- Man kan bruke metaparenteser for å gruppere symboler: $[\![\ldots]\!]$.

Det finnes (dessverre) mange variasjoner. BNF uten «$^{?}$», «*», «$^{+}$» og metaparenteser kalles gjerne **klassisk BNF**.

INF3110/4110

Ulike type språk

Språk som kan beskrives med en BNF, deles inn i følgende grupper:

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

  $$
  \begin{array}{lcl}
  \langle\text{binary number}\rangle & \rightarrow & \texttt{0} \mid \texttt{0}\ \langle\text{binary number}\rangle \mid \\
  & & \texttt{1} \mid \texttt{1}\ \langle\text{binary number}\rangle
  \end{array}
  $$

  Slike språk brukes til søk i Bash, Perl, Emacs, egrep, …

Hvorfor brukes regulære språk til søking?

❶ Det er enkelt å angi et ganske kraftig søkekriterium.

❷ Det er lett å lage en meget rask **automat** som sjekker lovlige uttrykk.

Dessverre har alle sin variant av regulære uttrykk.

INF3110/4110

- Type 2-språk («kontekst-frie») har bare et 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.

INF3110/4110

Praktiske konsekvenser

Type 0-språk er «sterkere» enn type-1-språk osv i det at de kan uttrykke mer.

Uttrykk på formen

$$\{ a^n b^n \mid n \geq 1 \}$$

kan ikke uttrykkes med en regulær grammatikk.

```
a*(3+i)
```

Uttrykk à la

$$\{ wcw \mid w \in (a|b)^* \}$$

kan ikke uttrykkes med en kontekstfri grammatikk.

```
int a_min;
if (a_min < 0) ...
```


Ulike representasjon 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

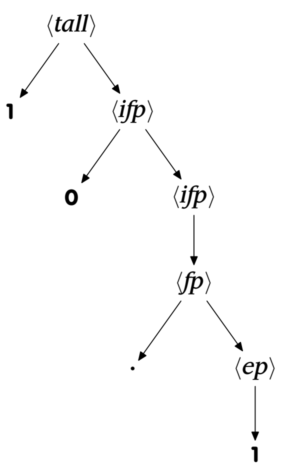
Representasjon 1: Klassisk BNF

$\langle \text{tall} \rangle \rightarrow$ 0 $\langle \text{fp} \rangle$ | 1 $\langle \text{ifp} \rangle$
$\langle \text{ifp} \rangle \rightarrow$ 1 $\langle \text{ifp} \rangle$ | 0 $\langle \text{ifp} \rangle$ | $\langle \text{fp} \rangle$
$\langle \text{fp} \rangle \rightarrow$ ϵ | . $\langle \text{ep} \rangle$
$\langle \text{ep} \rangle \rightarrow$ 0 | 1 | 0 $\langle \text{ep} \rangle$ | 1 $\langle \text{ep} \rangle$

(Symbolet «ϵ» 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 $\langle \text{tall} \rangle$ som regulært uttrykk blir

$$\langle \text{tall} \rangle \rightarrow \llbracket 0 \mid 1 \; \llbracket 0 \mid 1 \rrbracket^* \rrbracket \; \llbracket . \; \llbracket 0 \mid 1 \rrbracket^+ \rrbracket^?$$

Notasjon

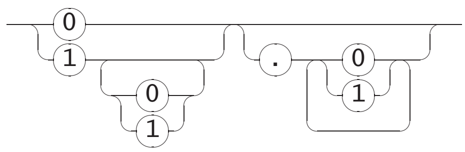
Selv om essensen er den samme, kan notasjonen for regulære uttrykk variere fra ett program til et annet. I Perl skriver vi

```
if (/^(0|1(0|1)*)(\.(0|1)+)?$/) {
    print "OK:   ";
} else {
    print "Feil: ";
}
```

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



Hvordan sjekke regulære uttrykk?

Det er ikke alltid enkelt:

```
#! /store/bin/perl -w

while (<>) {
    chomp;
    if (/^\s*[01]+B\s*$/) {
        $kind = "binært";
    } elsif (/^\s*[0-7]+O\s*$/) {
        $kind = "oktalt";
    } elsif (/^\s*\d+D\s*$/) {
        $kind = "desimalt";
    } elsif (/^\s*(\d|[abcdefABCDEF])+H\s*$/) {
        $kind = "heksadesimalt";
    } else {
        $kind = "feilaktig";
    }

    print "Et $kind tall: $_\n";
}
exit 0;
```

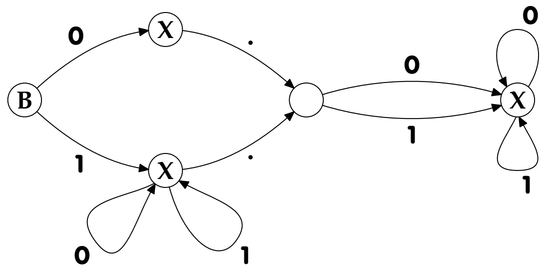
Eksempel:

```
    101000017O
    Et oktalt tall: 101000017O
    1234123412349999099c999D
    Et feilaktig tall: 1234123412349999099c999D
    1234123412349999099c999H
    Et heksadesimalt tall: 1234123412349999099c999H
```

Vi kan benytte en parser til sjekkingen, men det finnes en bedre mulighet: **FSM** («finite state machine» = endelig tilstandsmaskin).

Representasjon 4: Deterministisk FSM

Her definerer vi en FSM (ofte kalt **automat**) hvor tegnene fører oss fra én **tilstand** til neste.



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

Hvorfor er FSM-er så nyttige?

En FSM 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.

Søkealgoritme for FSM
Det finnes en enkel og meget rask søkealgoritme for en FSM lagret i matrisen re:

```
stat := 1;
while ⟨flere tegn igjen⟩ do begin
  c := ⟨neste tegn⟩;
  stat := re(stat,c);
end while;
if slutt(stat) then ⟨match funnet⟩
       else ⟨ingen match⟩;
```

Dette skjer altså i raske søkeprogrammer:

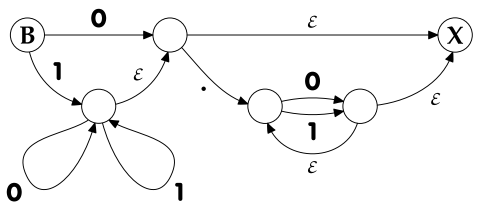
- Lag en FSM utifra det regulære uttrykket.
- Bruk søkeløkken over til å sjekke data mot det regulære uttrykket.

INF3110/4110

Hvordan lage en deterministisk FSM?

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

Først lages en ikke-deterministiske FSM:

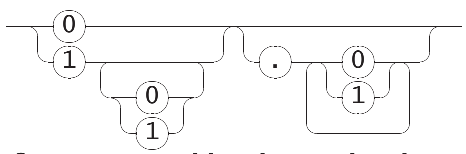


INF3110/4110

Hvordan lage en ikke-deterministisk FSM?

Ta utgangspunkt i jernbanediagrammet:

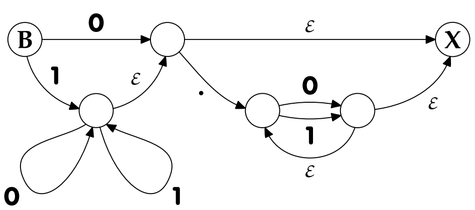
tall



❶ Hver «pens» blir til en node i den ikke-deterministiske FSM-en.

❷ Hvert sluttsymbol blir en merket kant. Noen får et tomt symbol (ϵ).

❸ Merk nodene hvor man skal begynne og slutte.



INF3110/4110

Hvordan bruke en ikke-deterministisk FSM?
En ikke-deterministisk FSM 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 (ϵ).

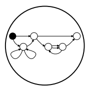
Imidlertid kan man lage en **deterministisk FSM** utifra en ikke-deterministisk.

INF3110/4110

Hvordan lage en deterministiske FSM-er

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

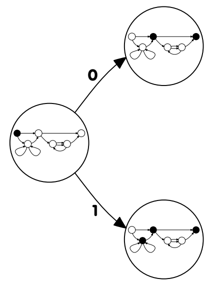
Lag en tilstand for disse nodene.



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

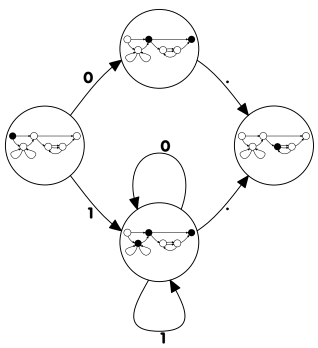
INF3110/4110

Fra hver tilstand følg kantene med ulike terminalsymboler (og tomme kanter). Utvid den deterministiske FSM-en med disse.



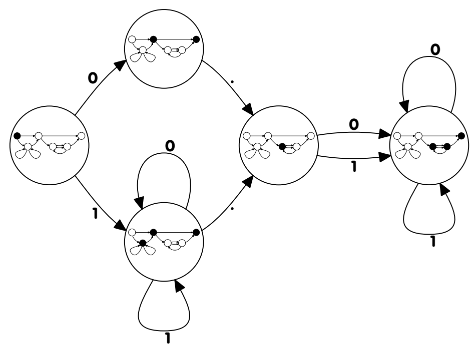
INF3110/4110

Fortsett med dette.

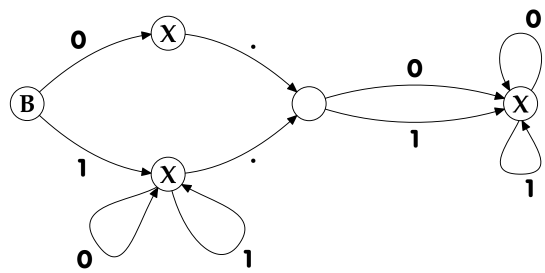


INF3110/4110

Til slutt har man den ferdige deterministiske FSM-en.



Vi ser at den er ekvivalent med den vi hadde på ark 19:



INF3110/4110