

Formella språk och syntaxanalys

Viggo Kann
viggo@kth.se

Allt du behöver veta om formella språk och syntaxanalys i en enda föreläsning!

- ❑ Alfabet, strängar, formella språk
- ❑ Ändliga automater
- ❑ Reguljära uttryck
- ❑ Uttrycksfullhet hos språkformalismer
- ❑ Kontextfria grammatiker, BNF, EBNF
- ❑ Syntaxträd
- ❑ Lexikal analys
- ❑ Syntaxanalys med rekursiv nedåkning

Formella språk

Ett *formellt språk* är en mängd *strängar* bestående av *tecken* från ett *alfabet*.

Exempel:

1. {xy, yxx, xyzzzy, zxy}
2. {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...}
3. {syntaktiskt korrekta program i Java 8}
4. {10, 1010, 101010, 10101010, ...}

Hur ett formellt språk definieras

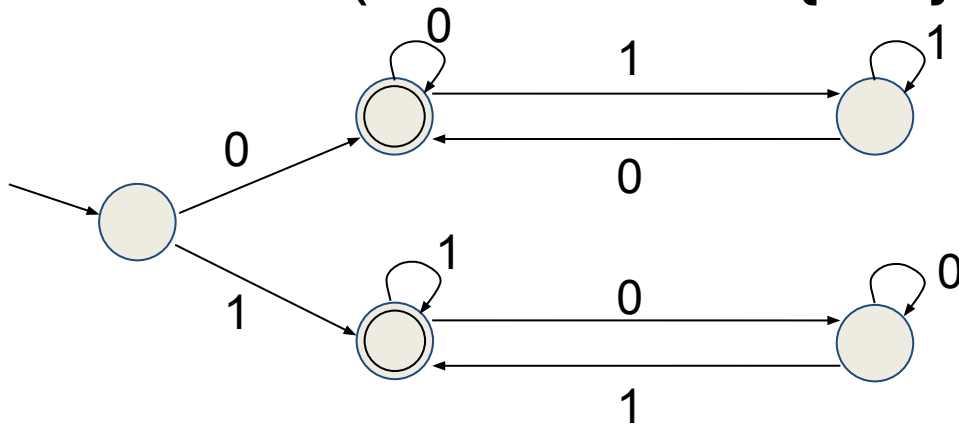
Det finns flera sätt att definiera ett språk L :

1. En lista med alla strängar som ingår i L
2. En grammatik bestående av regler som definierar hur strängarna i L ser ut
3. En algoritm som känner igen (accepterar) strängarna i språket, dvs, $A(s)=1 \Leftrightarrow s \in L$

Ändlig automat

(Finite state automaton, FSA)

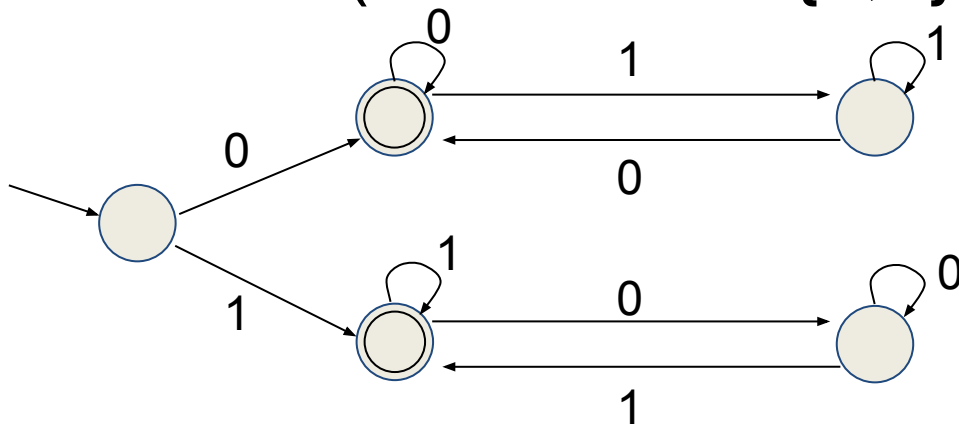
Exempel: Kolla om första och sista siffran i indata är samma (alfabetet är $\{0,1\}$)



Ändlig automat

(Finite state automaton, FSA)

Exempel: Kolla om första och sista siffran i indata är samma (alfabetet är $\{0,1\}$)

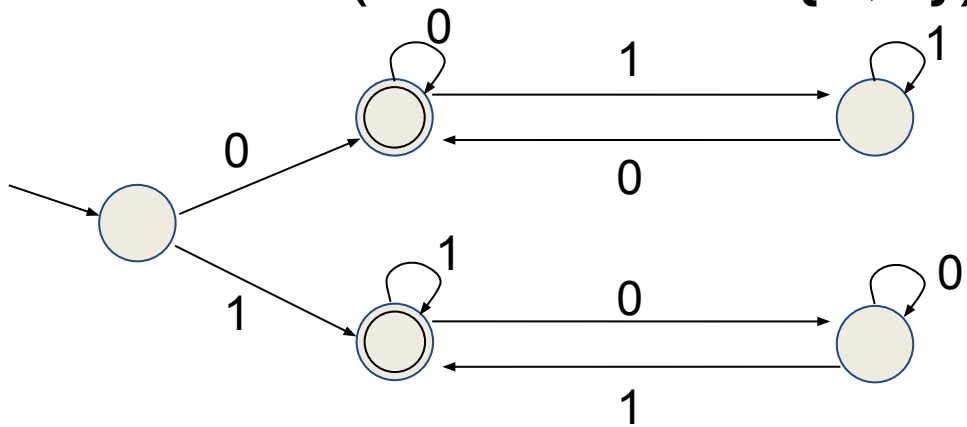


100101

Ändlig automat

(Finite state automaton, FSA)

Exempel: Kolla om första och sista siffran i indata är samma (alfabetet är $\{0,1\}$)



0011

En definition av en FSA består av

- Ett alfabet Σ
- En ändlig mängd tillstånd Q (ringar)
- Ett starttillstånd $q_0 \in Q$ (en ring som pekas på av en pil från rymden)
- En mängd accepterande tillstånd $F \subseteq Q$ (dubbla ringar)
- En övergångsfunktion $\delta: Q \times \Sigma \rightarrow Q$ (pilar)

Regler för en FSA

- ❑ Börja i starttillståndet
- ❑ En symbol läses i varje övergång (pil)
- ❑ Bara en övergång som är märkt med den symbol som står i tur att läsas kan följas
- ❑ När slutet på indata nåtts accepterar automaten om det aktuella tillståndet är ett accepterande tillstånd (dubbelring)

Deterministisk och ickedeterministisk

I en *deterministisk* FSA (DFSA eller DFA) är δ en *funktion*, dvs det finns aldrig två pilar från samma tillstånd märkta med samma symbol.

I en *ickedeterministisk* FSA (NFSA eller NFA) är δ en *relation*. Automaten accepterar inmatningen om det finns *någon* följd av övergångar som slutar i ett accepterande tillstånd.

Sats:

För varje NFSA finns det en DFSA som känner igen exakt samma språk.

Bevisidé: Låt varje delmängd av Q vara ett tillstånd i DFSA. Ett tillstånd (delmängd) B är accepterande om B innehåller minst ett accepterande tillstånd.

Eftersom varje DFSA uppfyller definitionen av en NFSA kan vi se att *beräkningsmodellerna DFSA och NFSA är lika uttrycksfulla.*

Reguljära uttryck (regular expression – RE)

Ett reguljärt uttryck är en grammatik som definierar ett språk med hjälp av operationerna *konkatenering*, *union* och (Kleene) *slutning*, samt parenteser för att ändra prioritetsordningen.

Union skrivs | and slutning skrivs *.

* har högst prioritet och union lägst.

Slutning betyder 0 eller flera stycken av det före *.

Exempel på reguljära uttryck

Reguljärt uttryck	Strängar i språket
$xy yxx xyzzy zxy$	$xy, yxx, xyzzy, zxy$
$10(10)^*$	$10, 1010, 101010, \dots$
$a^*(b c)$	$b, c, ab, ac, aab, aac, \dots$

Formell rekursiv definition av syntaxen för ett reguljärt uttryck

1. Tomma strängen ε är ett RE
2. Om a tillhör alfabetet så är a ett RE
3. Om R_1 och R_2 är RE så är (R_1R_2) ett RE (konkatenering)
4. Om R_1 och R_2 är RE så är $(R_1|R_2)$ ett RE (union)
5. Om R_1 är ett RE så är $(R_1)^*$ ett RE (slutning)

Reguljära språk

Ett språk som kan definieras med ett reguljärt uttryck är ett *reguljärt språk*.

RE är en delmängd av Unix reguljära uttryck (som också finns i bibliotek i Python, Java etc).

Varje Unix-RE kan skrivas om till ett RE.

T ex kan [0-9] skrivas som 0|1|2|3|4|5|6|7|8|9

Det betyder att Unix-RE inte är *uttrycksfullare* än RE.

Uttrycksfullhet (expressivity)

Grammatikformalismer kan vara mer eller mindre uttrycksfulla (med avseende på vilka språk dom kan definiera). *Chomskyhierarkin* definierar nivåer av uttrycksfullhet hos grammatikformalismer, där reguljära språk utgör en nivå.

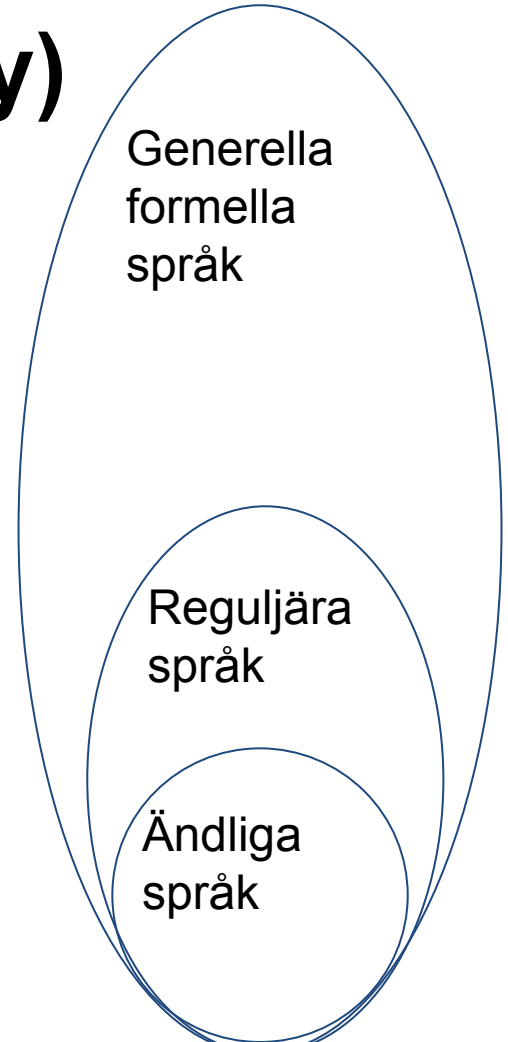
Större
uttrycks-
fullhet



Generella
formella
språk

Reguljära
språk

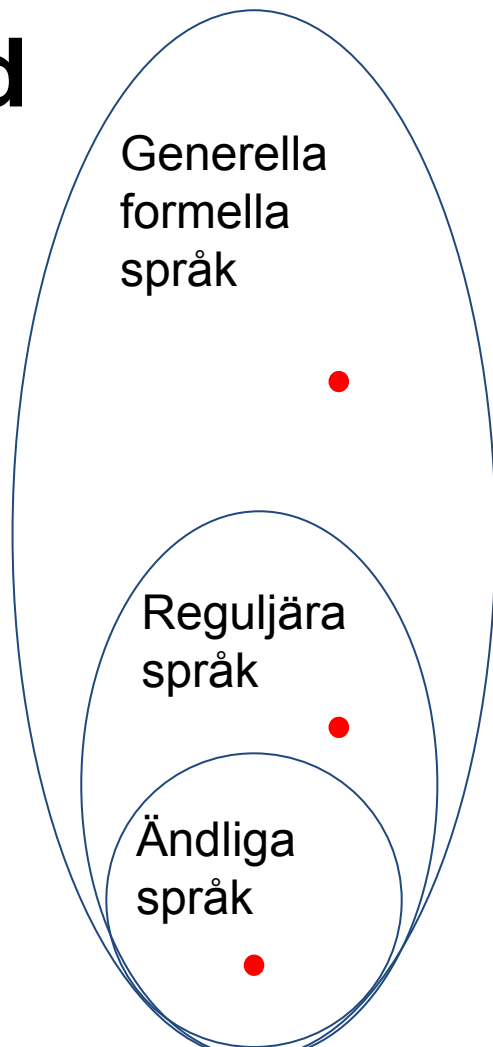
Ändliga
språk



Att jobba med på egen hand

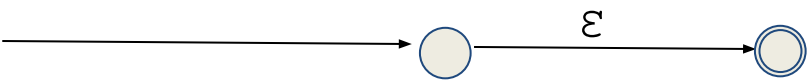
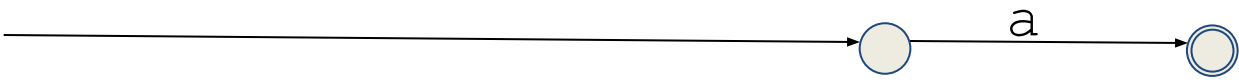
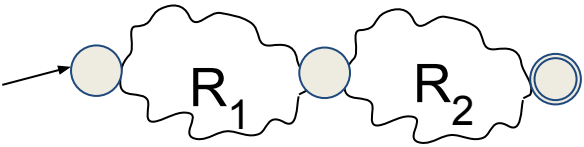
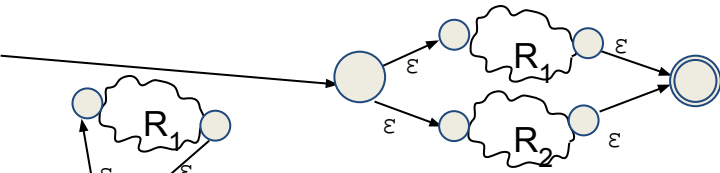
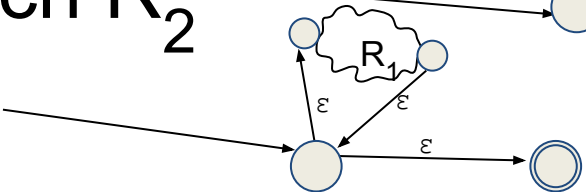
Beskriv tre formella språk,
ett för var och en av dom tre
nivåerna i bilden av
Chomskyhierarkin till höger!

Större
uttrycks-
fullhet



En NDFSA som motsvarar ett RE

Konstruktion av en NDFSA med hjälp av den rekursiva definitionen av RE:

1. Tomma strängen ε 
2. $a \in \Sigma$ 
3. Konkaterering av R_1 och R_2 
4. Unionen av R_1 och R_2 
5. Slutningen av R_1 

RE är lika uttrycksfull som DFSA

- ❑ Konstruera en NDFSA som accepterar exakt dom strängar som genereras av RE med den rekursiva definitionen (se förra bilden).
- ❑ Konstruera en DFSA som accepterar exakt dom strängar som accepteras av NDFSA (med delmängder som tillstånd).
- ❑ Det är också möjligt att, givet en DFSA, konstruera ett RE som genererar exakt dom strängar som accepteras av DFSA (beviset får inte plats här).

Kontextfri grammatik

(context-free grammar)

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

Kontextfri grammatik

Startsymbol (start symbol)

S → NP VP

VP → V | V NP

NP → Det N | Det N PP

Produktion (production)
eller omskrivningsregel
(rewriting rule)

PP → P NP

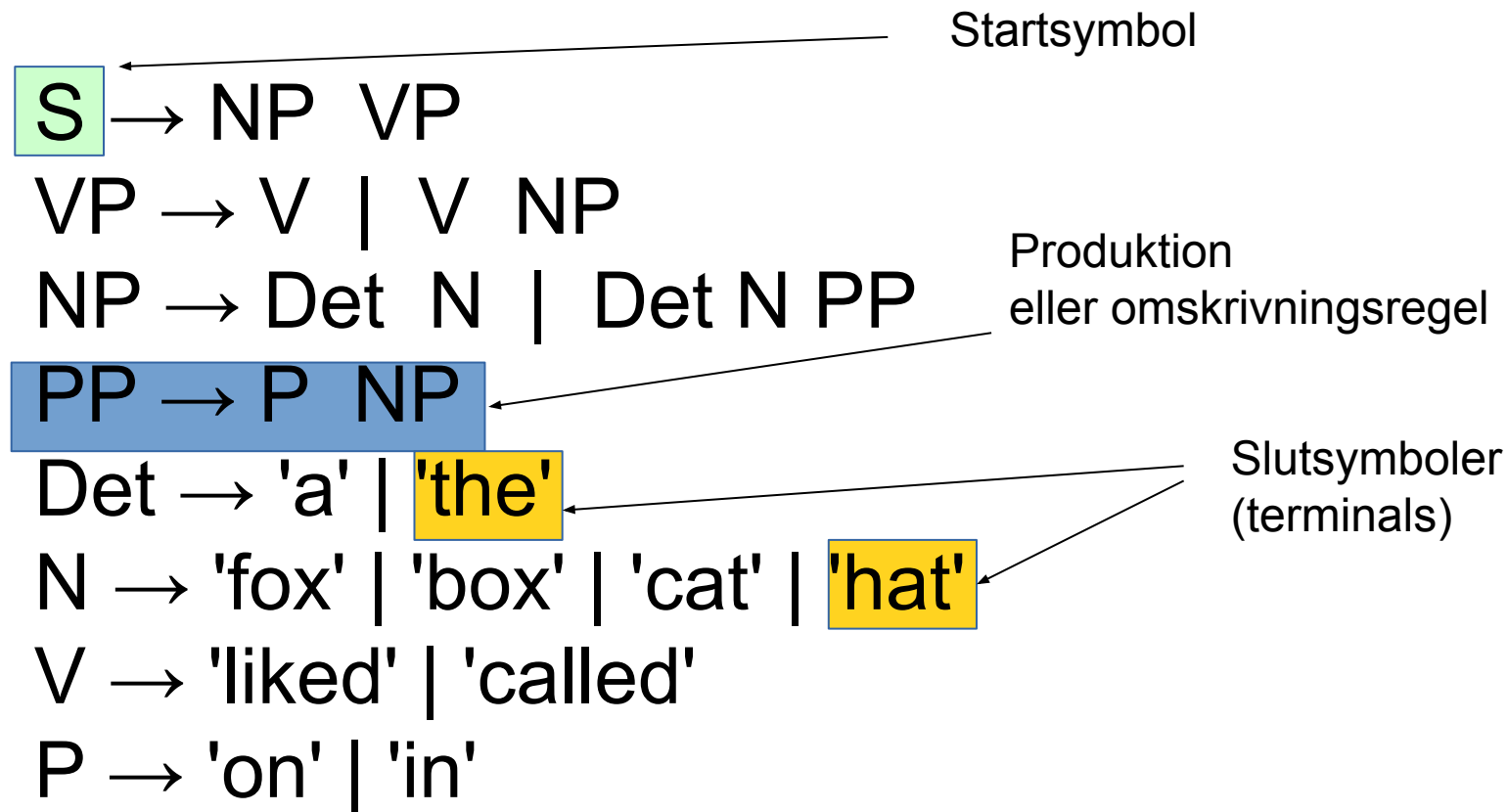
Det → 'a' | 'the'

N → 'fox' | 'box' | 'cat' | 'hat'

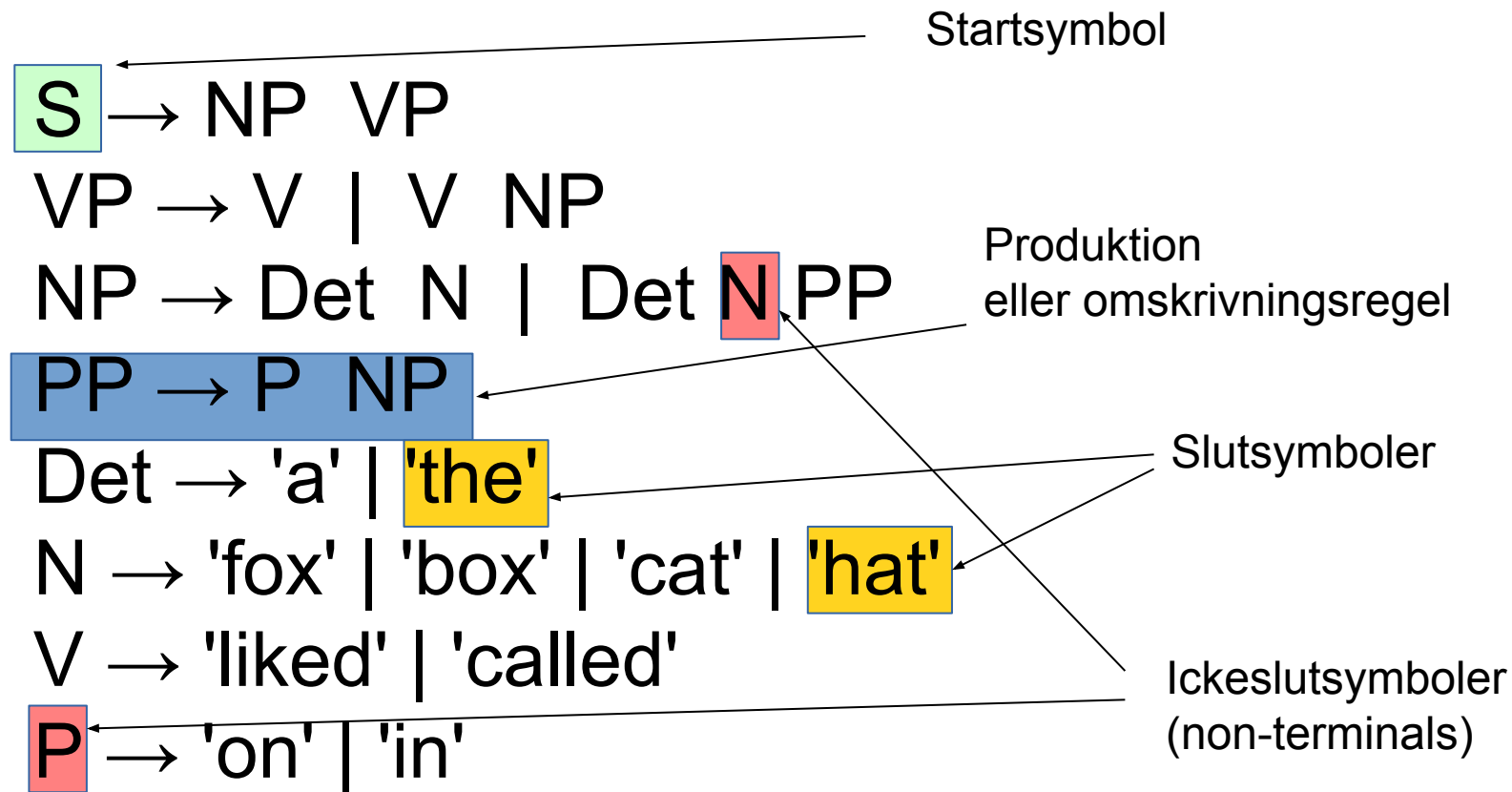
V → 'liked' | 'called'

P → 'on' | 'in'

Kontextfri grammatik



Kontextfri grammatik



Härledning (derivation)



$x \rightarrow y$

varje förekomst av x kan bytas mot y

$x \rightarrow y \mid z$

x kan bytas mot y eller z

Exempel: Frasen *the fox called* härleds från S

S \rightarrow NP VP

VP \rightarrow V | V NP

NP \rightarrow Det N | Det N PP

PP \rightarrow P NP

Det \rightarrow 'a' | 'the'

N \rightarrow 'fox' | 'box' | 'cat' | 'hat'

V \rightarrow 'liked' | 'called'

P \rightarrow 'on' | 'in'

S

the fox called

Exempel: Frasen *the fox called* härleds från S

S → NP VP

VP → V | V NP

NP → Det N | Det N PP

PP → P NP

Det → 'a' | 'the'

N → 'fox' | 'box' | 'cat' | 'hat'

V → 'liked' | 'called'

P → 'on' | 'in'

S
NP VP

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP

VP

Det N

VP

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP VP

Det N VP

the N VP

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP VP

Det N VP

the N VP

the fox VP

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP VP

Det N VP

the N VP

the fox VP

the fox V

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP VP

Det N VP

the N VP

the fox VP

the fox V

the fox called

Exempel: Frasen *the fox called* härleds från S

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP VP

Det N VP

the N VP

the fox VP

the fox V

the fox called

(Formellt) språk som motsvarar en grammatik

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

Alla fraser som kan
härledas från
startsymbolen
bildar *språket* som
genereras av
grammatiken

Vänsterhärledning/högerhärledning (left/right derivation)

Samma fras kan härledas på olika sätt.

- Om den *högraste* ickeslutsymbolen konsekvent väljs för omskrivning görs en *högerhärledning*.
- Om den *vänstraste* ickeslutsymbolen konsekvent väljs för omskrivning görs en *vänsterhärledning*.

Exempel: Högerhärledning av *the fox called*

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

S

NP

VP

NP

V

NP

called

Det

N

called

Det

fox called

the fox called

Syntaxanalysator (parser)

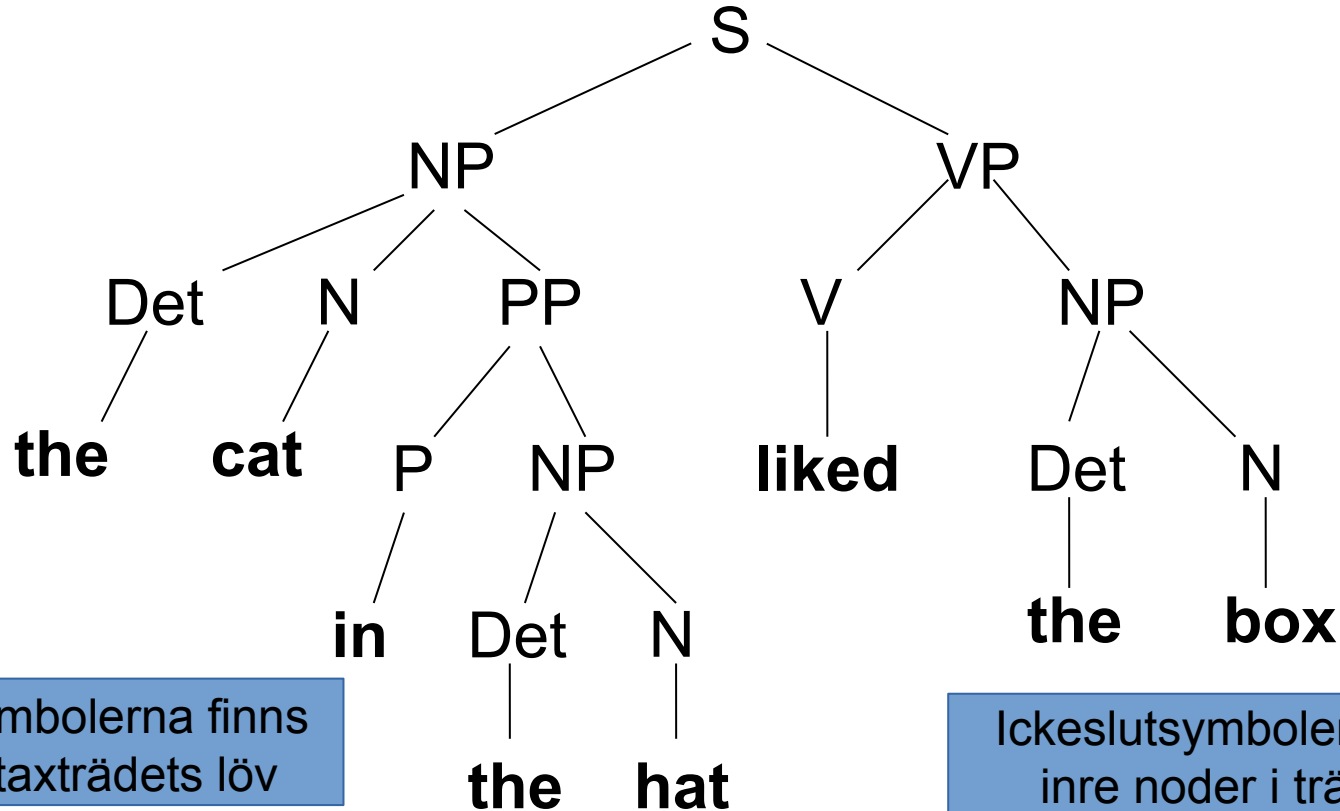
En syntaxanalysator visar hur en fras kan härledas i en given grammatik.

Uppifrån-ner-analysatorer (top-down parsers) använder vänsterhärledning.

Nerifrån-upp-analysatorer (bottom-up parsers) använder högerhärledning.

Resultatet av syntaxanalysen representeras ofta med ett *syntaxträd* (syntax tree).

Syntaxträd som härleder *the cat in the hat liked the box*



Att arbeta med på egen hand

$S \rightarrow NP \ VP$

$VP \rightarrow V \mid V \ NP$

$NP \rightarrow Det \ N \mid Det \ N \ PP$

$PP \rightarrow P \ NP$

$Det \rightarrow 'a' \mid 'the'$

$N \rightarrow 'fox' \mid 'box' \mid 'cat' \mid 'hat'$

$V \rightarrow 'liked' \mid 'called'$

$P \rightarrow 'on' \mid 'in'$

Konstruera ett
syntaxträd för
frasen

*a fox called the cat
in a box*

BNF – Backus Naur Form

Syntax för kontextfria grammatiker, utvecklad av Backus (för Fortran) och Naur (för Algol), kan skrivas på ett amerikanskt tangentbord. Vanligt sätt att definiera programspråk.

→ skrivs i BNF som $::=$

| används som *eller* mellan högerled för samma vänsterled

Ickeslutsymboler omges av $\langle \dots \rangle$

Exempel:

BNF-grammatik för taluttryck

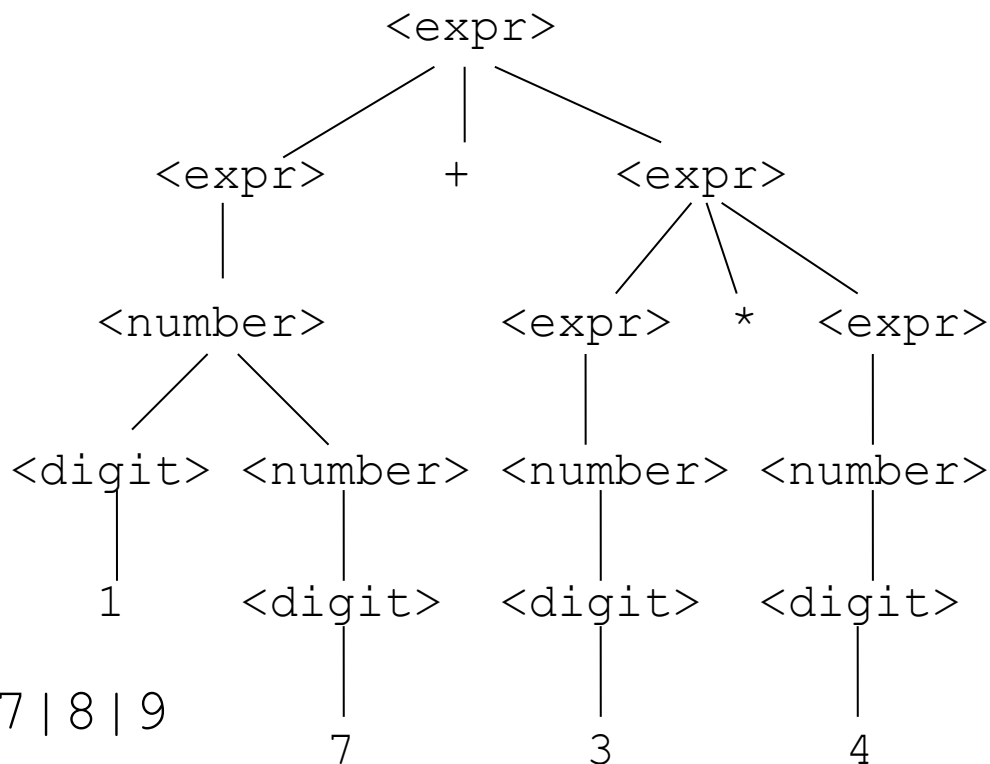
```
<expr> ::= <expr> + <expr>
        | <expr> - <expr>
        | <expr> * <expr>
        | <expr> / <expr>
        | <expr> ^ <expr>
        | - <expr>
        | ( <expr> )
        | <number>
        <number> ::= <digit>
                    | <digit><number>
        <digit> ::= 0 | 1
                    | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```


Exempel: syntaxträd för $17+3*4$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle - \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle * \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle / \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle ^ \langle \text{expr} \rangle$
 | $- \langle \text{expr} \rangle$
 | $(\langle \text{expr} \rangle)$
 | $\langle \text{number} \rangle$

$\langle \text{number} \rangle ::= \langle \text{digit} \rangle$
 | $\langle \text{digit} \rangle \langle \text{number} \rangle$

$\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$



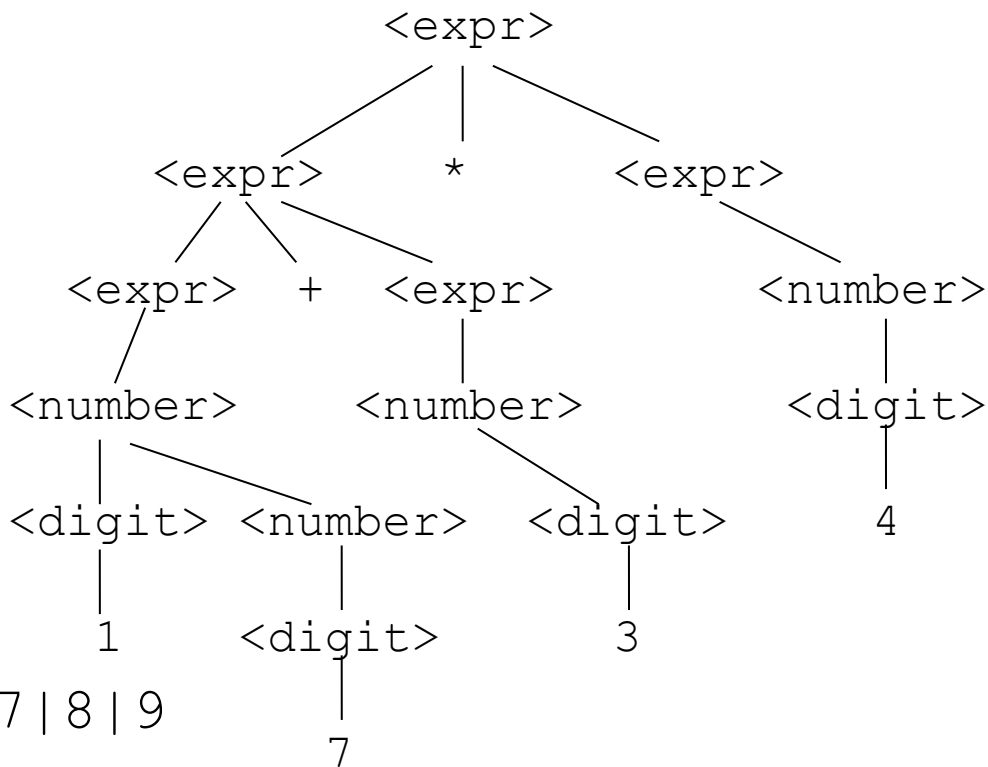
Flertydigt: alternativt syntaxträd för $17+3*4$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle - \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle * \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle / \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle ^ \langle \text{expr} \rangle$
 | $- \langle \text{expr} \rangle$
 | $(\langle \text{expr} \rangle)$
 | $\langle \text{number} \rangle$

$\langle \text{number} \rangle ::= \langle \text{digit} \rangle$

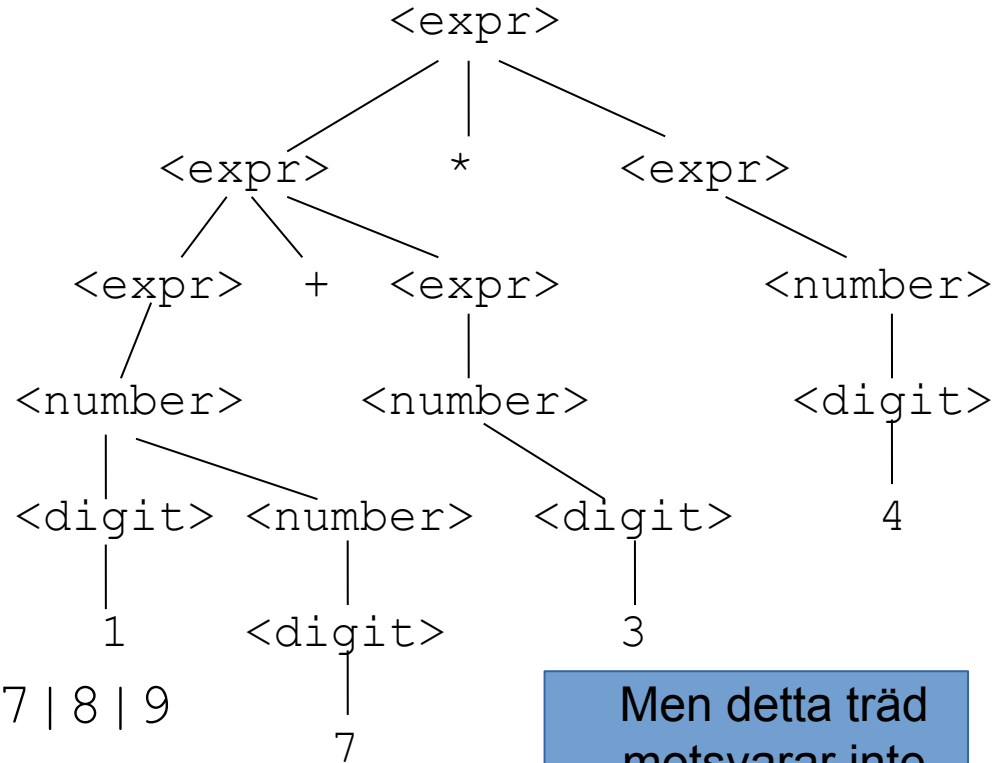
 | $\langle \text{digit} \rangle \langle \text{number} \rangle$

$\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$



Flertydigt: alternativt syntaxträd för $17+3*4$

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle - \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle * \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle / \langle \text{expr} \rangle$
 | $\langle \text{expr} \rangle ^ \langle \text{expr} \rangle$
 | $- \langle \text{expr} \rangle$
 | $(\langle \text{expr} \rangle)$
 | $\langle \text{number} \rangle$
 $\langle \text{number} \rangle ::= \langle \text{digit} \rangle$
 | $\langle \text{digit} \rangle \langle \text{number} \rangle$
 $\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$



Men detta träd
motsvarar inte
semantiken!

Att göra en grammatik entydig

Ibland kan man genom att införa nya ickeslutsymboler göra om en flertydig grammatik så att den blir *entydig* (unambiguous).

Om detta inte är möjligt är grammatiken *ohjälpligt flertydig* (inherently ambiguous).

Entydig grammatik för taluttryck

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$

| $\langle \text{expr} \rangle + \langle \text{term} \rangle$

| $\langle \text{expr} \rangle - \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= \langle \text{fact} \rangle$

| $\langle \text{term} \rangle * \langle \text{fact} \rangle$

| $\langle \text{term} \rangle / \langle \text{fact} \rangle$

$\langle \text{fact} \rangle ::= \langle \text{prim} \rangle$

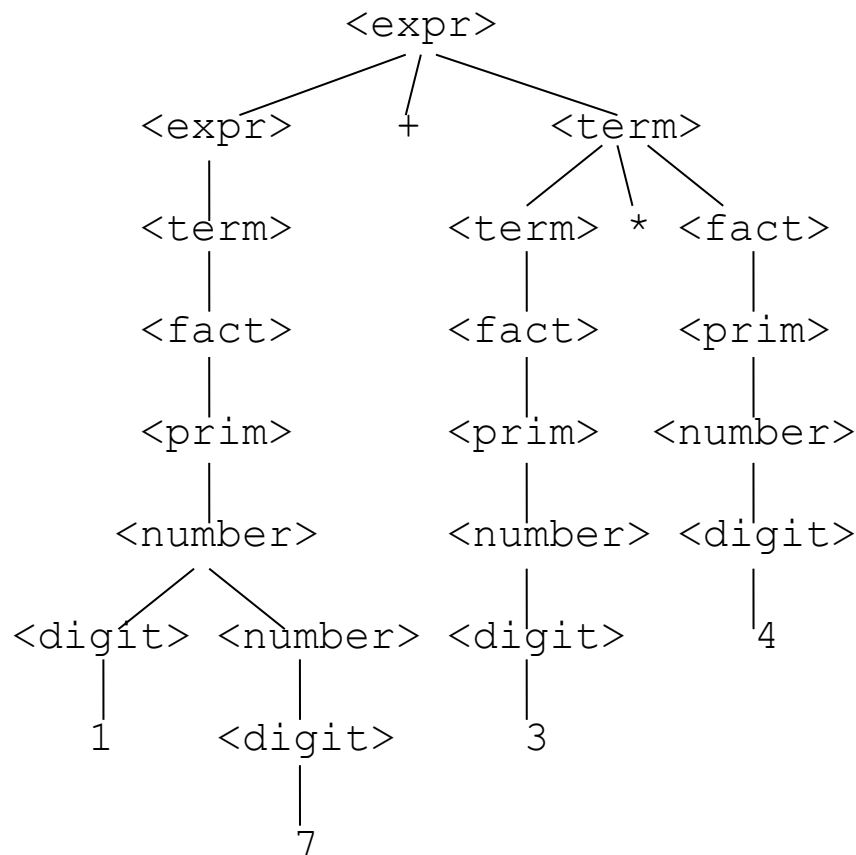
| $\langle \text{prim} \rangle ^ \langle \text{fact} \rangle$

$\langle \text{prim} \rangle ::= \langle \text{number} \rangle$

| $- \langle \text{prim} \rangle$

| $(\langle \text{expr} \rangle)$

$\langle \text{number} \rangle ::= \text{...som tidigare...}$



EBNF – Extended Backus Naur Form

(...) används för att gruppera

`<expr> ::= <expr> (+ | -) <term> | <term>`

[...] betyder 0 eller 1 förekomst av ...

`<expr> ::= [<expr> (+ | -)] <term>`

{...} betyder 0 eller flera förekomster av ...

`<number> ::= <digit> { <digit> }`

EBNF bekvämt men inte uttrycksfullare än BNF

EBNF – Att arbeta med på egen hand

(...) används för att gruppera

[...] betyder 0 eller 1 förekomst av ...

{...} betyder 0 eller flera förekomster av ...

Hur bevisar man att EBNF inte är mer uttrycksfullt än vanlig BNF?

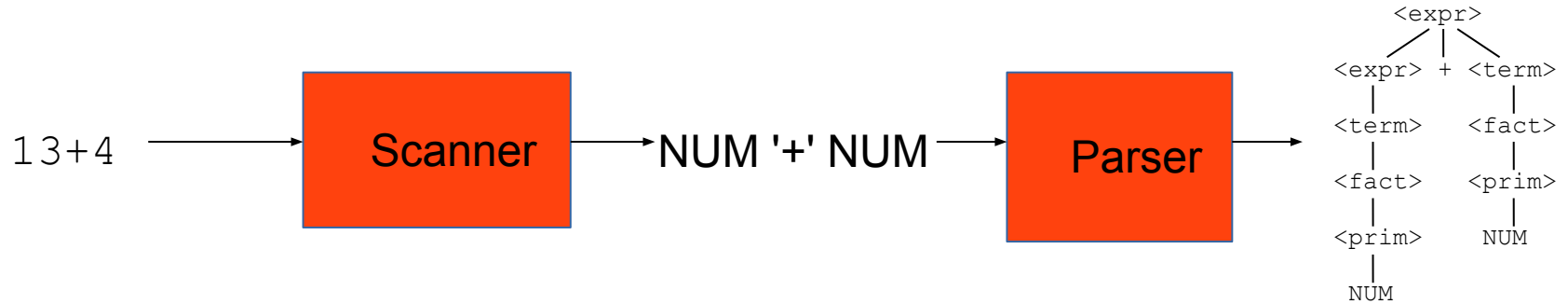
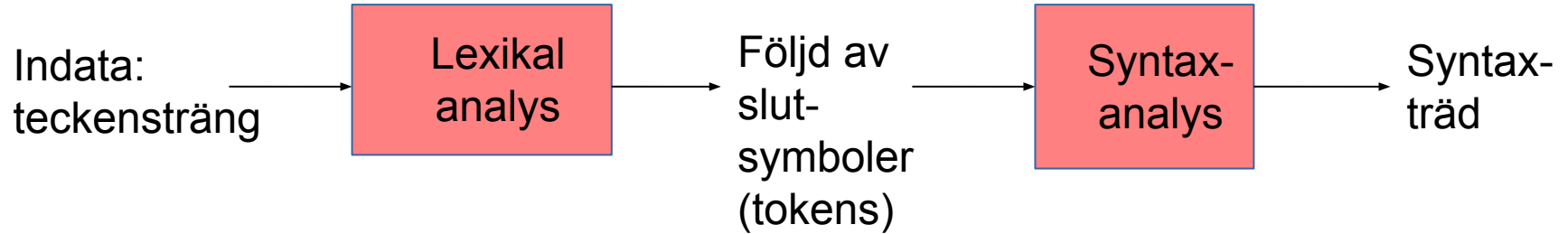
BNF-grammatik och motsvarande EBNF-grammatik

```
<expr> ::= <term>
          | <expr> + <term>
          | <expr> - <term>
<term> ::= <fact>
          | <term> * <fact>
          | <term> / <fact>
<fact> ::= <prim>
          | <prim> ^ <fact>
<prim> ::= <number>
          | - <prim>
          | (<expr>)
<number> ::= <digit>
            | <digit> <number>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<expr> ::= [ <expr> ( + | - ) ] <term>
<term> ::= [ <term> ( * | / ) ] <fact>
<fact> ::= <prim> [ ^ <fact> ]
<prim> ::= <number> | - <prim>
           | ' ( ' <expr> ' ) '
<number> ::= <digit> { <digit> }
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Betydligt kompaktare i EBNF
men kanske mer svårläst?

Parsningsprocessen



Förutsägande syntaxanalysator (predictive parser)

Idé:

- Härled frasen uppifrån-ner (dvs från startsymbolen till slutsymbolerna)
- Välj vilken regel (dvs högerled) som ska tillämpas genom att tjuvkika på nästa symbol i indata

Exempel:

```
Prim ::= Number  
      | '-' Prim  
      | '(' Expr ')'
```

Indata, tecken: - **4711**

Indata, slutsymboler: - Number

Nästa symbol i indata: -

Välj därför regeln `Prim ::= '-' Prim`

Problem 1: Högerled som börjar med samma symbol

```
Fact ::= Prim  
      | Prim '^' Fact
```

Lösning: Faktorisera ut det gemensamma prefixet och skapa en ny regel med resten.

```
Fact ::= Prim Rest  
Rest ::= ε  
      | '^' Fact
```

Problem 2: Vänsterrekursion

$$\begin{aligned} \text{Expr} &::= \text{Term} \\ &| \text{Expr} \text{ '+' } \text{Term} \\ &| \text{Expr} \text{ '-' } \text{Term} \end{aligned}$$

Lösning:

- ▣ Separera vänsterrekursiva högerled från övriga
- ▣ Skapa en ny ickeslutsymbol H
- ▣ Lägg till H på slutet av alla icke-vänsterrekursiva högerled
- ▣ Låt högerleden till H vara dom vänsterrekursiva högerleden omgjorda till högerrekursiva

Problem 2: Vänsterrekursion, exempel

$$\begin{aligned}\text{Expr} &::= \text{Term} \\ &\quad | \text{Expr } '+' \text{ Term} \\ &\quad | \text{Expr } '-' \text{ Term}\end{aligned}$$

\Rightarrow

$$\begin{aligned}\text{Expr} &::= \text{Term } H \\ H &::= \varepsilon \\ &\quad | '+' \text{ Term } H \\ &\quad | '-' \text{ Term } H\end{aligned}$$

Rekursiv medåkningsanalys (recursive descent parsing)

Uppifrån-neranalysator implementerad med funktioner konstruerade från grammatiken.

1. En lexikalanalysatorfunktion `Match(sym)` som kollar att nästa symbol i indata är `sym` och läser in efterföljande symbol i `nextToken`.
2. En funktion för varje ickeslutsymbol som analyserar indata genom att anropa funktioner i den ordning motsvarande symboler förekommer i högerledet

Rekursiv medåkning, exempel

Fact ::= Prim Rest

Rest ::= ε

| '^' Fact

implementeras som:

Fact() : Prim(); Rest()

Rest() : **if** nextToken='^' :

Match('^'); Fact()

Vi kan kombinera dessa till en enda funktion!


```
Expr() : Term()
        while nextToken in { '+', '-' } :
            Match(nextToken); Term()

Term() : Fact()
        while nextToken in { '*', '/' } :
            Match(nextToken); Fact()

Fact() : Prim()
        if nextToken='^' :
            Match('^'); Fact()
```

```
Prim() : case nextToken of:  
    Number: Match(Number)  
    '-' : Match('-',  
        Prim()  
    '  
    '(' : Match('(',  
        Expr()  
        Match(')')
```

casio.c - en enkel miniräknare med operationerna + - * / ^ implementerad med rekursiv medåkning

```
typedef enum /* slutsymboler */
{
    START, ERROR, NUMBERSYM, ENDSYM,
    ADDSYM = '+', MINUSSYM = '-',
    MULSYM = '*', DIVSYM = '/',
    EXPSYM = '^', PRINTSYM = ';',
    LEFTPARSYM = '(', RIGHTPARSYM = ')'
} TokenValue;
```

```
/* nästa slutsymbol i indata */
TokenValue nextToken;
```

```
/* semantiskt värde för NUMBERSYM */
double numberValue;
```

```
/* Match kollar att token matchar
indata och läser in nytt nextToken */
TokenValue Match(TokenValue token);
```

```
/* funktioner för ickeslutsymboler:
*/
```

```
double prim(void);
double fact(void);
double term(void);
double expr(void);
void start(void);
```

```
int main(void) /* huvudprogram */
{ Match(START); start(); }
```

```
double prim(void)
{ double e;
  switch (nextToken) {
    case NUMBERSYM:
      Match(NUMBERSYM);
      return numberValue;
    case MINUSSYM:
      Match(MINUSSYM);
      return -prim();
    case LEFTPARSYM:
      Match(LEFTPARSYM);
      e = expr();
      if (nextToken != RIGHTPARSYM)
        return error(" förväntades.");
      Match(RIGHTPARSYM);
      return e;
    default:
      return error("En faktor förväntades.");
  }
}
```

```
double fact(void)
{ double left = prim();
  if (nextToken != EXPSYM)
    return left;
  Match(EXPSYM);
  return pow(left, fact());
}
```

```

double term(void)
{ double d, left = fact();
  while (1)
    switch (nextToken) {
      case MULSYM:
        Match(MULSYM);
        left *= fact();
        break;
      case DIVSYM:
        Match(DIVSYM);
        d = fact();
        if (d == 0)
          return error("Division med 0.");
        left /= d;
        break;
      default:
        return left;
    }
}

```

```

double expr(void)
{ double left = term();
  while (1)
    switch (nextToken) {
      case ADDSYM:
        Match(ADDSYM);
        left += term();
        break;
      case MINUSSYM:
        Match(MINUSSYM);
        left -= term();
        break;
      default:
        return left;
    }
}

```

```

TokenValue Match(TokenValue token)
{ int ch;
  if (token != nextToken && token != START) {
    error("Felaktig symbol.\n"); return ERROR;
  }
  while (1) {
    ch = getchar();
    if (isdigit(ch) || ch == '.') {
      ungetc(ch, stdin);
      if (scanf("%lf", &numberValue) == 1) return (nextToken = NUMBERSYM);
      else ch = getchar();
    }
    switch (ch) {
      case EOF: return (nextToken = ENDSYM);
      case '\t':
      case ' ': break;
      case '\n': return (nextToken = PRINTSYM);
      case '^': return (nextToken = EXPSYM);
      case '*': return (nextToken = MULSYM);
      case '/': return (nextToken = DIVSYM);
      case '+': return (nextToken = ADDSYM);
      case '-': return (nextToken = MINUSSYM);
      case '(': return (nextToken = LEFTPARSYM);
      case ')': return (nextToken = RIGHTPARSYM);
      default: fprintf(stderr, "Felaktigt tecken: %c\n", ch);
    }
  }
}

```