

DD1361

Programmeringsparadigm

Formella Språk & Syntaxanalys  
Föreläsning 3

Per Austrin

2016-11-10

# Snabb repetition

Huvudkoncept hittils:

## *Formellt språk*

- en mängd strängar

## *Reguljära språk*

- den klass av formella språk som kan beskrivas med reguljära uttryck eller ändliga automater (DFA:er)
- viktigt: reguljära uttryck och automater lika kraftfulla, kan beskriva samma saker

## *Grammatiker*

- verktyg för att beskriva språk med enkla rekursiva definitioner

Idag

## Lexikal analys

Härledninggar och syntaxträd

Rekursiv medåkning

Omskrivning av grammatiker

# Grammatik för aritmetiska uttryck

Låt oss skriva en grammatik för aritmetiska uttryck med heltal, operatorerna  $+$ ,  $-$ ,  $*$ ,  $/$ , och parenteser.

# Grammatik för aritmetiska uttryck

Låt oss skriva en grammatik för aritmetiska uttryck med heltal, operatorerna  $+$ ,  $-$ ,  $*$ ,  $/$ , och parenteser.

Möjlig grammatik:

$$\begin{aligned} \text{Expr} &\rightarrow \text{Number} \mid \\ &\quad \text{Expr} + \text{Expr} \mid \\ &\quad \text{Expr} - \text{Expr} \mid \\ &\quad \text{Expr} * \text{Expr} \mid \\ &\quad \text{Expr} / \text{Expr} \mid \\ &\quad (\text{Expr}) \end{aligned}$$
$$\text{Number} \rightarrow \text{Digit} \mid \text{Digit Number}$$
$$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

**Icke-slutsymboler:**

**Expr** (representerar uttryck)

**Number** (representerar heltal)

**Digit** (representerar siffror)

# Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

```
Expr → Number |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

```
Number → Digit | Digit Number
```

```
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

# Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

Reglerna för vad som är ett tal känns lite som ett bihang som vi var tvungna att ha med för att göra grammatiken fullständig.

```
Expr → Number |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

```
Number → Digit | Digit Number
```

```
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

# Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

Reglerna för vad som är ett tal känns lite som ett bihang som vi var tvungna att ha med för att göra grammatiken fullständig.

Att känna igen tal är ju väldigt enkelt (kan t.ex. göras med det reguljära uttrycket  $[0-9]^+$ ), är lite overkill att använda grammatik för det.

```
Expr → Number |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

```
Number → Digit | Digit Number
```

```
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

Detta är *teckensekvensen*

'3', '7', '8', '\*', '2', '3', '2', '\*', '(', '5', '8', '2', '-', '0', '1', ')'

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

Detta är *teckensekvensen*

'3', '7', '8', '\*', '2', '3', '2', '\*', '(', '5', '8', '2', '-', '0', '1', ')'

Vi pre-processar detta till den nya sekvensen

Number, '\*', Number, '\*', '(', Number, '-', Number, ')'

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

Detta är *teckensekvensen*

'3', '7', '8', '\*', '2', '3', '2', '\*', '(', '5', '8', '2', '-', '0', '1', ')'

Vi pre-processar detta till den nya sekvensen

Number, '\*', Number, '\*', '(', Number, '-', Number, ')'

Elementen i den nya sekvensen kallas för *tokens*.

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

Detta är *teckensekvensen*

'3', '7', '8', '\*', '2', '3', '2', '\*', '(', '5', '8', '2', '-', '0', '1', ')'

Vi pre-processar detta till den nya sekvensen

Number, '\*', Number, '\*', '(', Number, '-', Number, ')'

Elementen i den nya sekvensen kallas för *tokens*.

Några tokens är helt enkelt enstaka tecken från indata-strängen, men andra består av delsträngar från indata-strängen som getts en typ (i det här fallet "Number")

# Pre-processa heltal

**Idé:** *pre-processa* indata-strängen för att ta hand om de delar som utgör "enkla" beståndsdelar av grammatiken

**Exempel:** låt oss titta på strängen "378\*232\*(582-01)"

Detta är *teckensekvensen*

'3', '7', '8', '\*', '2', '3', '2', '\*', '(', '5', '8', '2', '-', '0', '1', ')'

Vi pre-processar detta till den nya sekvensen

Number, '\*', Number, '\*', '(', Number, '-', Number, ')'

Elementen i den nya sekvensen kallas för *tokens*.

Några tokens är helt enkelt enstaka tecken från indata-strängen, men andra består av delsträngar från indata-strängen som getts en typ (i det här fallet "Number")

Vi betraktar denna *tokensekvens* som ett indata att parse

# Åter till grammatiken

Vad händer med vår grammatik?

$\text{Expr} \rightarrow \text{Number} \mid$

$\text{Expr} + \text{Expr} \mid$

$\text{Expr} - \text{Expr} \mid$

$\text{Expr} * \text{Expr} \mid$

$\text{Expr} / \text{Expr} \mid$

$(\text{Expr})$

$\text{Number} \rightarrow \text{Digit} \mid \text{Digit Number}$

$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



# Åter till grammatiken

Vad händer med vår grammatik?

$\text{Expr} \rightarrow \text{Number} \mid$

$\text{Expr} + \text{Expr} \mid$

$\text{Expr} - \text{Expr} \mid$

$\text{Expr} * \text{Expr} \mid$

$\text{Expr} / \text{Expr} \mid$

$(\text{Expr})$

~~$\text{Number} \rightarrow \text{Digit} \mid \text{Digit Number}$~~   
 ~~$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$~~

# Åter till grammatiken

Vad händer med vår grammatik?

$\text{Expr} \rightarrow \text{Number} \mid$

$\text{Expr} + \text{Expr} \mid$

$\text{Expr} - \text{Expr} \mid$

$\text{Expr} * \text{Expr} \mid$

$\text{Expr} / \text{Expr} \mid$

$(\text{Expr})$

~~$\text{Number} \rightarrow \text{Digit} \mid \text{Digit} \text{Number}$~~   
 ~~$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$~~

Man kan säga att vi har “uppgraderat” (eller nedgraderat?) “Number” till en slut-symbol istället för en icke-slutsymbol

# Lexikal analys

*Lexikal analys* är processen att transformera en indatasträng (sekvens av tecken) till en sekvens av tokens.

Syften:

1. Abstrahera bort smådetaljer ur grammatiken.
2. Städa bort irrelevanta delar av indata
  - Kommentarer i programmeringsspråk
  - Whitespace

Vi vill antagligen att "12+5" och "12 + 5" ska behandlas likadant, men att "1 2+5" ska behandlas annorlunda

# Lexikal analys

*Lexikal analys* är processen att transformera en indatasträng (sekvens av tecken) till en sekvens av tokens.

Syften:

1. Abstrahera bort smådetaljer ur grammatiken.
2. Städa bort irrelevanta delar av indata
  - Kommentarer i programmeringsspråk
  - Whitespace

Vi vill antagligen att "12+5" och "12 + 5" ska behandlas likadant, men att "1 2+5" ska behandlas annorlunda

**Tumregel:** om något kan beskrivas med ett enkelt reguljärt uttryck så är det (oftast) bättre att betrakta det som ett token istället för att skriva grammatik-regler för det.

T.ex. tal, beskrevs enkelt som  $[0-9]^+$

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
  where
    -- beräkna differensen
    x = alpha-beta
```

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Vad är lämpliga val av tokens i Haskell?

Förslag:

**Where/Let/Derives/etc** alla keywords i språket

**Equal/Plus/Minus/Times/etc** operatorer

**Int/Double/etc** tal av olika typer

**Name** Variabel/funktionsnamn

(och många fler)

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Detta är en kommentar så vi ignorerar den



# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, **Equal**

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, **Where**



# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where

Detta är en kommentar så vi ignorerar den

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name,  
Equal

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name,  
Equal, Name

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name,  
Equal, Name, **Minus**

# Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name,  
Equal, Name, Minus, Name

Idag

Lexikal analys

Härledning och syntaxträd

Rekursiv medåkning

Omskrivning av grammatiker

# Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?



# Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

## Exempel

- Är  $4*(5+3)$  ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

# Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

## Exempel

- Är  $4*(5+3)$  ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

Men ofta har ju strängarna i språken en mening – en *semantik* – som vi vill kunna analysera

# Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

## Exempel

- Är  $4*(5+3)$  ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

Men ofta har ju strängarna i språken en mening – en *semantik* – som vi vill kunna analysera

## Exempel

- Vad är värdet av  $4*(5+3)$ ?
- Vad händer när mitt java-program körs?

# Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

# Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

**Exempel:** Låt oss härleda "4\*(5+3)",  
dvs token-sekvensen  
Num, '\*', '(', Num, '+', Num, ')'

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

# Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

**Exempel:** Låt oss härleda “4\*(5+3)”, dvs token-sekvensen

Num, '\*', '(', Num, '+', Num, ')'

## Härledning:

$\text{Expr} \rightarrow \text{Expr} * \text{Expr}$   
 $\rightarrow \text{Num} * \text{Expr}$   
 $\rightarrow \text{Num} * (\text{Expr})$   
 $\rightarrow \text{Num} * (\text{Expr} + \text{Expr})$   
 $\rightarrow \text{Num} * (\text{Num} + \text{Expr})$   
 $\rightarrow \text{Num} * (\text{Num} + \text{Num})$

$\text{Expr} \rightarrow \text{Num} \mid$   
 $\text{Expr} + \text{Expr} \mid$   
 $\text{Expr} - \text{Expr} \mid$   
 $\text{Expr} * \text{Expr} \mid$   
 $\text{Expr} / \text{Expr} \mid$   
 $(\text{Expr})$

# Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

**Exempel:** Låt oss härleda “4\*(5+3)”, dvs token-sekvensen

Num, '\*', '(', Num, '+', Num, ')'

$$\begin{aligned} \text{Expr} &\rightarrow \text{Num} \mid \\ &\text{Expr} + \text{Expr} \mid \\ &\text{Expr} - \text{Expr} \mid \\ &\text{Expr} * \text{Expr} \mid \\ &\text{Expr} / \text{Expr} \mid \\ &(\text{Expr}) \end{aligned}$$

## Härledning:

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\ &\rightarrow \text{Num} * \text{Expr} \\ &\rightarrow \text{Num} * (\text{Expr}) \\ &\rightarrow \text{Num} * (\text{Expr} + \text{Expr}) \\ &\rightarrow \text{Num} * (\text{Num} + \text{Expr}) \\ &\rightarrow \text{Num} * (\text{Num} + \text{Num}) \end{aligned}$$

Varje steg använder en av produktionsreglerna, och vi landar till slut i indata-sekvensen

# Syntaxträd

Det bästa sättet att representera härledningarna på är med *syntaxträd*



# Syntaxträd

Det bästa sättet att representera härledningarna på är med *syntaxträd*

Härledning

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\ &\rightarrow \text{Num} * \text{Expr} \\ &\rightarrow \text{Num} * (\text{Expr}) \\ &\rightarrow \text{Num} * (\text{Expr} + \text{Expr}) \\ &\rightarrow \text{Num} * (\text{Num} + \text{Expr}) \\ &\rightarrow \text{Num} * (\text{Num} + \text{Num}) \end{aligned}$$

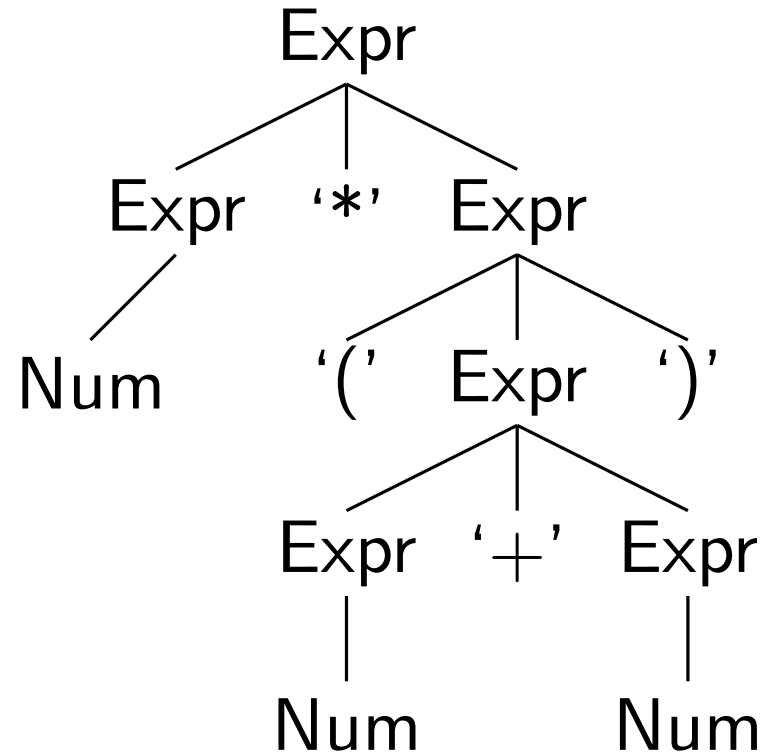
# Syntaxträd

Det bästa sättet att representera härledningarna på är med *syntaxträd*

Härledning

$\text{Expr} \rightarrow \text{Expr} * \text{Expr}$   
 $\rightarrow \text{Num} * \text{Expr}$   
 $\rightarrow \text{Num} * (\text{Expr})$   
 $\rightarrow \text{Num} * (\text{Expr} + \text{Expr})$   
 $\rightarrow \text{Num} * (\text{Num} + \text{Expr})$   
 $\rightarrow \text{Num} * (\text{Num} + \text{Num})$

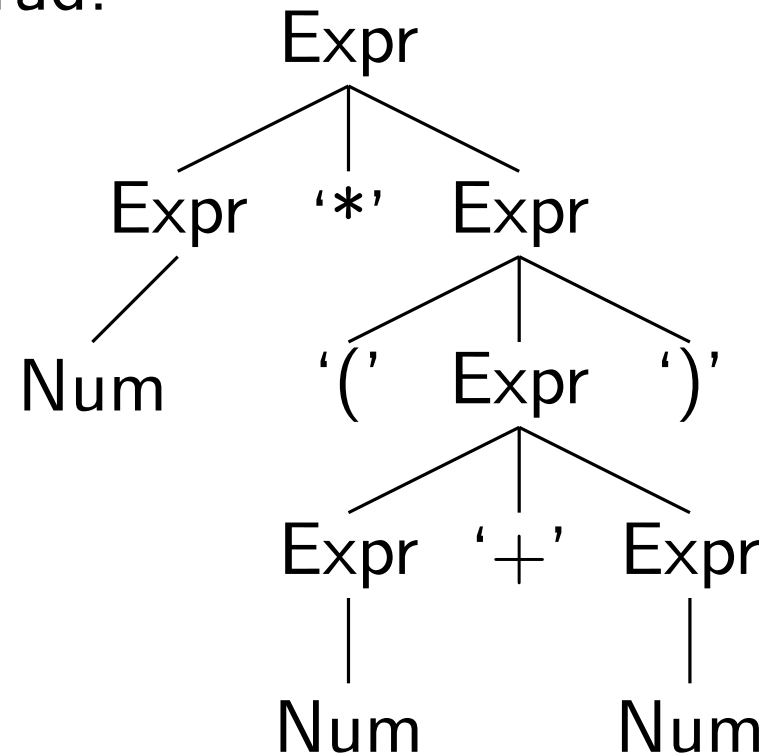
Visualiserat som syntaxträd



# Semantik från härledning/syntaxträd

Uttryck:  $4*(5+3)$

Syntaxträd:

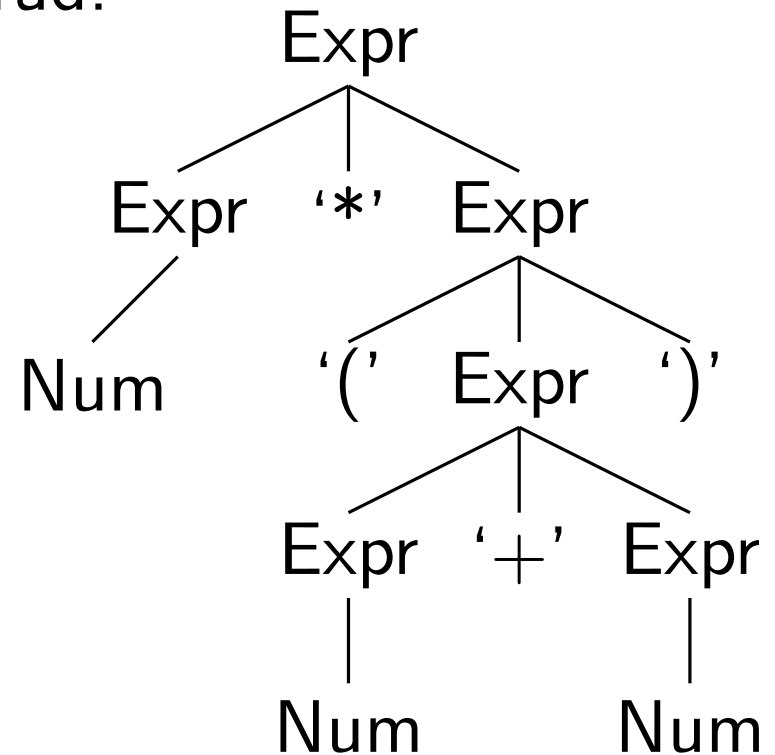


# Semantik från härledning/syntaxträd

Uttryck:  $4*(5+3)$

Syntaxträd:

För att kunna beräkna värdet av uttrycket behöver vi ju veta exakt vilka tal som de olika Num-elementen var.



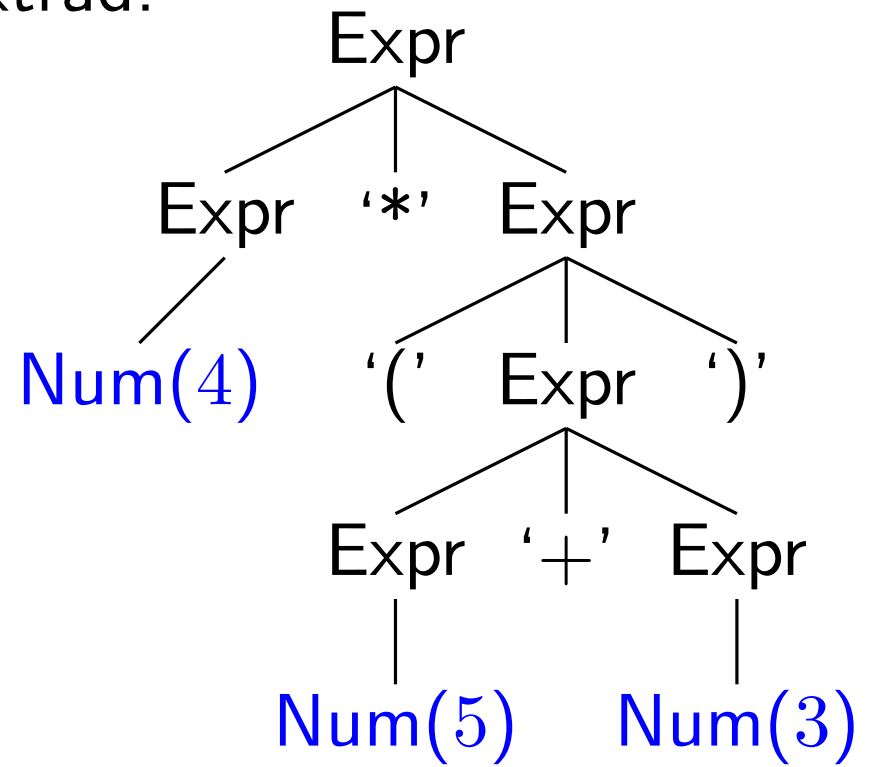
# Semantik från härledningingar/syntaxträd

Uttryck:  $4*(5+3)$

Syntaxträd:

För att kunna beräkna värdet av uttrycket behöver vi ju veta exakt vilka tal som de olika Num-elementen var.

Varje Num-token "taggas" med semantisk data som talar om vilket tal det representerar



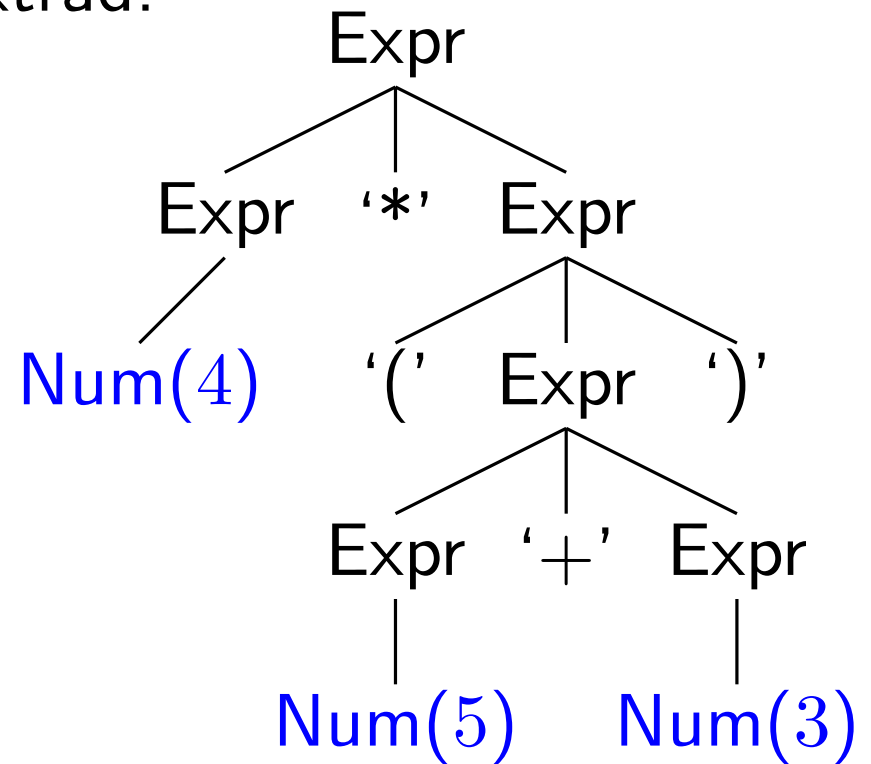
# Semantik från härledning/syntaxträd

Uttryck:  $4*(5+3)$

Syntaxträd:

För att kunna beräkna värdet av uttrycket behöver vi ju veta exakt vilka tal som de olika Num-elementen var.

Varje Num-token "taggas" med semantisk data som talar om vilket tal det representerar



Vi kan nu beräkna värdet med en enkel sökning i trädet.

```
if leafnode: value = token.value()
else if plusnode: value = left.value() + right.value()
else if mulnode: value = left.value() * right.value()
etc...
```

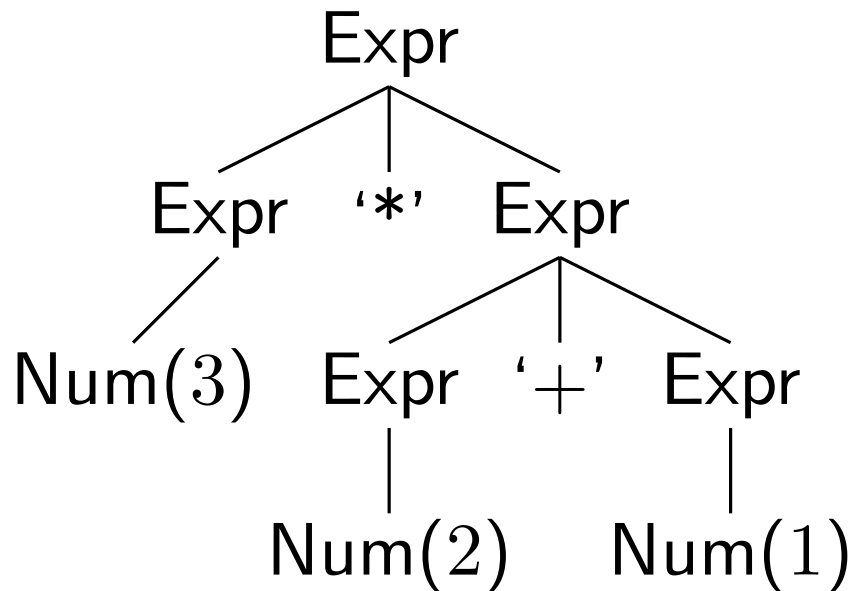
# Syntaxträd, exempel 2

Uttryck:  $3*2+1$

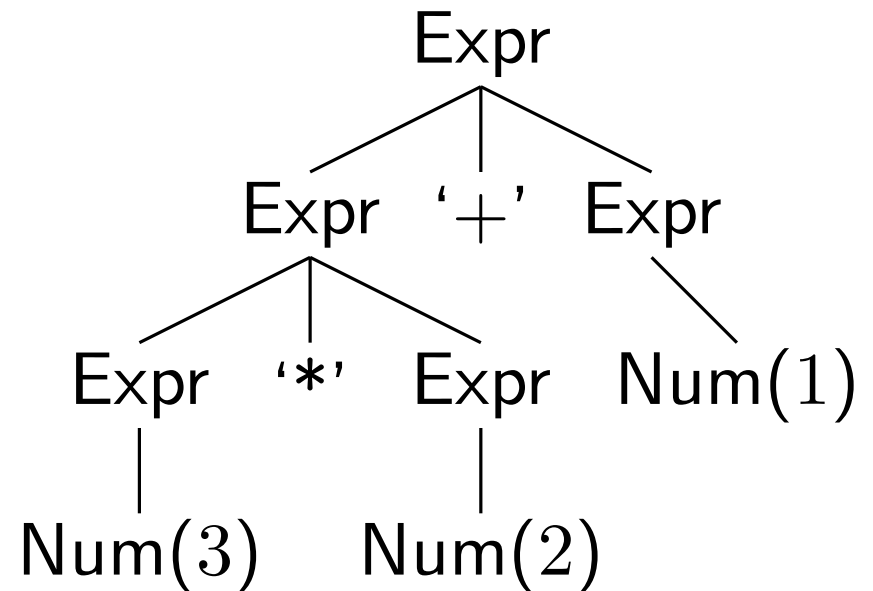
## Syntaxträd, exempel 2

Uttryck:  $3*2+1$

Anna hävdar att det har följande syntaxträd:



Bengt hävdar att det har följande syntaxträd:



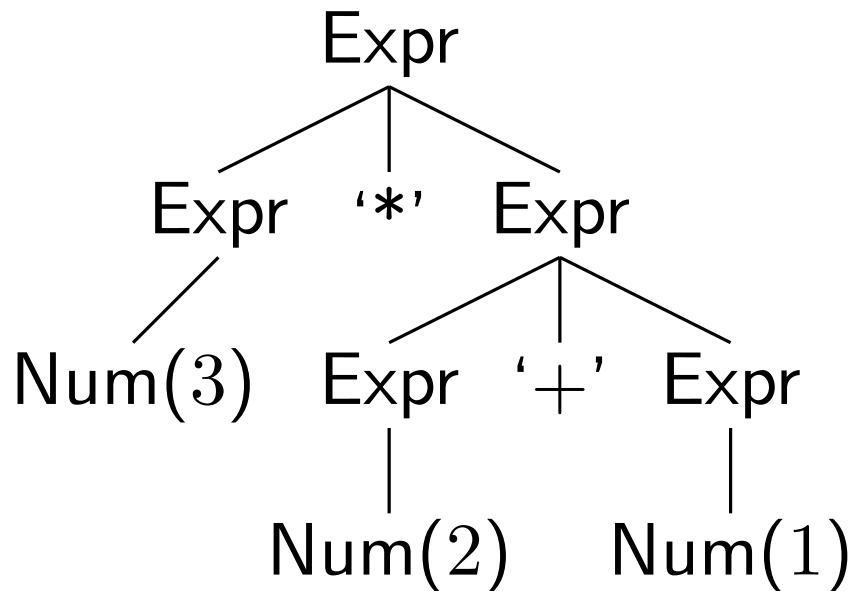
**Vem har rätt?**



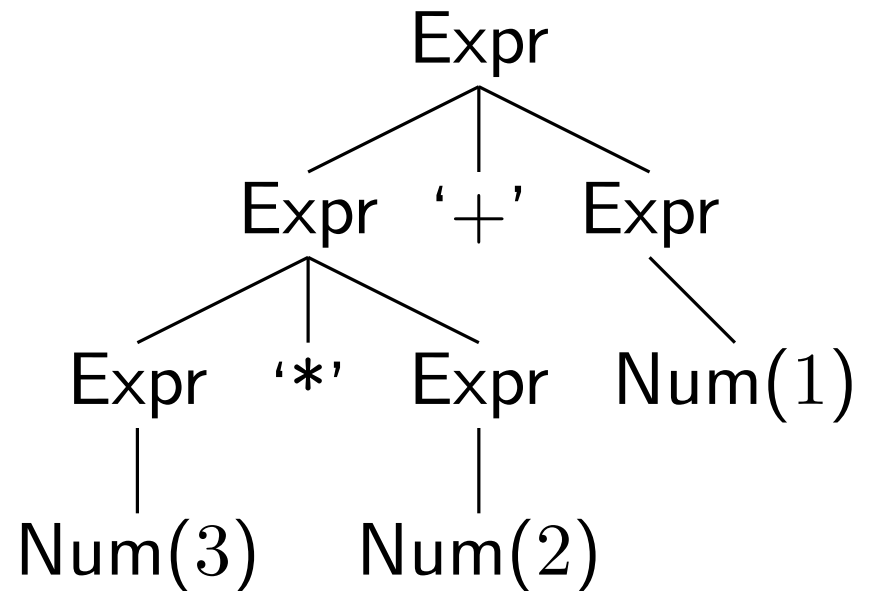
## Syntaxträd, exempel 2

Uttryck:  $3*2+1$

Anna hävdar att det har följande syntaxträd:



Bengt hävdar att det har följande syntaxträd:



**Vem har rätt?**

Båda är korrekta syntaxträd för uttrycket enligt grammatiken!

# Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

# Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

# Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar:  $3 \cdot (2 + 1) = 9$  vs.  $(3 \cdot 2) + 1 = 7$

# Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar:  $3 \cdot (2 + 1) = 9$  vs.  $(3 \cdot 2) + 1 = 7$

Att en grammatik är tvetydig spelar ingen roll för syntaxen (ändrar inte vad som är giltiga strängar), men gör att det finns flera olika semantiska tolkningar av strängarna i språket.

# Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar:  $3 \cdot (2 + 1) = 9$  vs.  $(3 \cdot 2) + 1 = 7$

Att en grammatik är tvetydig spelar ingen roll för syntaxen (ändrar inte vad som är giltiga strängar), men gör att det finns flera olika semantiska tolkningar av strängarna i språket.

Sådan dubbeltydighet kan vara charmig i naturliga språk, men antagligen inte önskvärd i ett programmeringsspråk...

Idag

Lexikal analys

Härledninggar och syntaxträd

Rekursiv medåkning

Omskrivning av grammatiker

# Rekursiv medåkning

Rekursiv medåkning är en “meta-algoritm” för att konstruera en parser för en grammatik.



# Rekursiv medåkning

Rekursiv medåkning är en “meta-algoritm” för att konstruera en parser för en grammatik.

Huvudidé:

- en funktion per icke-slutsymbol, som är ansvarig för att parse den icke-slutsymbolen.
- tittar på nästa token i indata och väljer produktionsregel baserat på det
- sedan rekursiva anrop för att parse de olika delarna av högerledet

# Grammatik för binära träd

BinTree  $\rightarrow$  Leaf LParen Number RParen |

Branch LParen BinTree Comma BinTree RParen

# Grammatik för binära träd

BinTree  $\rightarrow$  Leaf LParen Number RParen |

Branch LParen BinTree Comma BinTree RParen

## Slutsymboler:

Leaf: strängen "leaf"

Branch: strängen "branch"

Number:  $[0-9]^+$

LParen, RParen, Comma: parenteser och kommatecken

# Grammatik för binära träd

BinTree  $\rightarrow$  Leaf LParen Number RParen |

Branch LParen BinTree Comma BinTree RParen

## Slutsymboler:

Leaf: strängen "leaf"

Branch: strängen "branch"

Number: [0-9]+

LParen, RParen, Comma: parenteser och kommatecken

## Exempel:

branch(branch(leaf(17), leaf(42)), leaf(5))

# Grammatik för binära träd

BinTree  $\rightarrow$  Leaf LParen Number RParen |

Branch LParen BinTree Comma BinTree RParen

## Slutsymboler:

Leaf: strängen "leaf"

Branch: strängen "branch"

Number: [0-9]+

LParen, RParen, Comma: parenteser och kommatecken

## Exempel:

branch(branch(leaf(17), leaf(42)), leaf(5))

## Efter lexikal analys:

Branch, LParen, Branch, LParen, Leaf, LParen, Number, RParen, Comma, Leaf, LParen, Number, RParen, RParen, Comma, Leaf, LParen, Number, RParen, RParen

# Rekursiv medåkning för binära träd

BinTree  $\rightarrow$  Leaf(Number) | Branch(BinTree, BinTree)

# Rekursiv medåkning för binära träd

`BinTree`  $\rightarrow$  `Leaf(Number)` | `Branch(BinTree, BinTree)`

Bara en icke-slutsymbol, vi ska ha en funktion `BinTree` som är ansvarig för att parse `BinTree`

# Rekursiv medåkning för binära träd

BinTree  $\rightarrow$  Leaf(Number) | Branch(BinTree, BinTree)

Bara en icke-slutsymbol, vi ska ha en funktion BinTree som är ansvarig för att parse BinTree

```
ParseTree BinTree() {
    Token t = NextToken();
    if (t.type == Leaf) {
        if (NextToken().type != LParen) throw SyntaxError();
        Token Num = NextToken();
        if (Num.type != Number) throw SyntaxError();
        if (NextToken().type != RParen) throw SyntaxError();
        return new LeafNode(Num.data);
    } else if (t.type == Branch) {
        if (NextToken().type != LParen) throw SyntaxError();
        ParseTree left = BinTree();
        if (NextToken().type != Comma) throw SyntaxError();
        ParseTree right = BinTree();
        if (NextToken().type != RParen) throw SyntaxError();
        return new BranchNode(left, right);
    }
}
```



# Rekursiv medåkning för binära träd

BinTree L (N L) | R L (P: T R: T) Tree)

Bara en  
ansvarig

Fullständig Java-implementation av lexikal  
analys och rekursiv medåknings-parser för  
binära träd finns på kurshemsidan under  
"kursmaterial" för dagens föreläsning

ee som är

ParseTree B

```
Token t = NextToken();
if (t.type == Leaf) {
    if (NextToken().type != LParen) throw SyntaxError();
    Token Num = NextToken();
    if (Num.type != Number) throw SyntaxError();
    if (NextToken().type != RParen) throw SyntaxError();
    return new LeafNode(Num.data);
} else if (t.type == Branch) {
    if (NextToken().type != LParen) throw SyntaxError();
    ParseTree left = BinTree();
    if (NextToken().type != Comma) throw SyntaxError();
    ParseTree right = BinTree();
    if (NextToken().type != RParen) throw SyntaxError();
    return new BranchNode(left, right);
}
}
```

## Rekursiv medåkning, exempel 2

Grammatik med slutsymbolerna b, c, d, f, p, och z:

$$\text{Start} \rightarrow c \text{ Cat} \mid d \text{ Dog}$$
$$\text{Cat} \rightarrow p \mid z \mid f \text{ Dog} \mid \text{Start Start}$$
$$\text{Dog} \rightarrow p \mid z \mid b \text{ Cat} \mid \text{Start}$$

## Rekursiv medåkning, exempel 2

Grammatik med slutsymbolerna b, c, d, f, p, och z:

$$\text{Start} \rightarrow c \text{ Cat} \mid d \text{ Dog}$$
$$\text{Cat} \rightarrow p \mid z \mid f \text{ Dog} \mid \text{Start Start}$$
$$\text{Dog} \rightarrow p \mid z \mid b \text{ Cat} \mid \text{Start}$$

Konstruktion av rekursiv medåknings-parser är **nästan mekanisk** från grammatiken:

## Rekursiv medåkning, exempel 2

Grammatik med slutsymbolerna b, c, d, f, p, och z:

$$\text{Start} \rightarrow c \text{ Cat} \mid d \text{ Dog}$$
$$\text{Cat} \rightarrow p \mid z \mid f \text{ Dog} \mid \text{Start Start}$$
$$\text{Dog} \rightarrow p \mid z \mid b \text{ Cat} \mid \text{Start}$$

Konstruktion av rekursiv medåknings-parser är **nästan mekanisk** från grammatiken:

- Det finns tre icke-slutsymboler “Start”, “Cat”, “Dog”, alltså ska det finnas tre funktioner Start(), Cat(), och Dog()

## Rekursiv medåkning, exempel 2

Grammatik med slutsymbolerna b, c, d, f, p, och z:

$$\text{Start} \rightarrow c \text{ Cat} \mid d \text{ Dog}$$
$$\text{Cat} \rightarrow p \mid z \mid f \text{ Dog} \mid \text{Start Start}$$
$$\text{Dog} \rightarrow p \mid z \mid b \text{ Cat} \mid \text{Start}$$

Konstruktion av rekursiv medåknings-parser är **nästan mekanisk** från grammatiken:

- Det finns tre icke-slutsymboler “Start”, “Cat”, “Dog”, alltså ska det finnas tre funktioner `Start()`, `Cat()`, och `Dog()`
- Icke-slutsymbolen “Cat” har fyra produktionsregler, i funktionen `Cat()` ska vi välja en av dessa fyra baserat på nästa tillgängliga indata-tecken.

## Rekursiv medåkning, exempel 2

För att implementera `Cat()` kollar vi på reglerna för `Cat`:

`Cat`  $\rightarrow$  `p` | `z` | `f Dog` | `Start Start`

## Rekursiv medåkning, exempel 2

För att implementera `Cat()` kollar vi på reglerna för `Cat`:

`Cat`  $\rightarrow$  `p` | `z` | `f Dog` | `Start Start`

```
function Cat()  
    char next = peekChar();
```

## Rekursiv medåkning, exempel 2

För att implementera `Cat()` kollar vi på reglerna för `Cat`:

`Cat`  $\rightarrow$  `p` | `z` | `f Dog` | `Start Start`

```
function Cat()  
    char next = peekChar();  
    if (next == 'p') // Cat, regel 1  
        eatChar(); // gå förbi 'p'
```



## Rekursiv medåkning, exempel 2

För att implementera `Cat()` kollar vi på reglerna för `Cat`:

`Cat` → `p` | `z` | `f Dog` | `Start Start`

```
function Cat()  
    char next = peekChar();  
    if (next == 'p')           // Cat, regel 1  
        eatChar();           // gå förbi 'p'  
    else if (next == 'z') // Cat, regel 2  
        eatChar();           // gå förbi 'z'
```

## Rekursiv medåkning, exempel 2

För att implementera `Cat()` kollar vi på reglerna för `Cat`:

`Cat` → `p` | `z` | `f Dog` | `Start Start`

```
function Cat()
    char next = peekChar();
    if (next == 'p')           // Cat, regel 1
        eatChar();           // gå förbi 'p'
    else if (next == 'z')     // Cat, regel 2
        eatChar();           // gå förbi 'z'
    else if (next == 'f')     // Cat, regel 3
        eatChar();           // gå förbi 'f'
        Dog();               // ska nu komma en Dog
```

## Rekursiv medåkning, exempel 2

För att implementera Cat() kollar vi på reglerna för Cat:

Cat → p | z | f Dog | Start Start

```
function Cat()
    char next = peekChar();
    if (next == 'p')           // Cat, regel 1
        eatChar();           // gå förbi 'p'
    else if (next == 'z')     // Cat, regel 2
        eatChar();           // gå förbi 'z'
    else if (next == 'f')     // Cat, regel 3
        eatChar();           // gå förbi 'f'
        Dog();                // ska nu komma en Dog
    else                       // måste vara Cat, regel 4
        Start();              // första Start
        Start();              // andra Start
```

# Idag

Lexikal analys

Härledninggar och syntaxträd

Rekursiv medåkning

Omskrivning av grammatiker

# Tillbaka till tvetydighet

Åter till vår tvetydiga grammatik för aritmetiska uttryck.

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

# Tillbaka till tvetydighet

Åter till vår tvetydiga grammatik för aritmetiska uttryck.

## Exempel:

$3*2+1$

$5-2*3/2+3*(5-3)$

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

# Tillbaka till tvetydighet

Åter till vår tvetydiga grammatik för aritmetiska uttryck.

## Exempel:

$3*2+1$

$5-2*3/2+3*(5-3)$

*Vi vill att \* och / ska ha högre prioritet än + och -*

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

# Tillbaka till tvetydighet

Åter till vår tvetydiga grammatik för aritmetiska uttryck.

## Exempel:

$3*2+1$

$5-2*3/2+3*(5-3)$

*Vi vill att \* och / ska ha högre prioritet än + och -*

Men detta finns inte med i grammatiken

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```



# Tillbaka till tvetydighet

Åter till vår tvetydiga grammatik för aritmetiska uttryck.

## Exempel:

$3*2+1$

$5-2*3/2+3*(5-3)$

*Vi vill att \* och / ska ha högre prioritet än + och -*

```
Expr → Num |  
      Expr + Expr |  
      Expr - Expr |  
      Expr * Expr |  
      Expr / Expr |  
      (Expr)
```

Men detta finns inte med i grammatiken

Ett mer detaljerat resonemang om hur aritmetiska uttryck ser ut:

- uttryck består av en eller flera termer (separerade av + eller -)
- termer består av en eller flera faktorer (separerade av \* eller /)
- faktorer är antingen ett tal, eller uttryck omgivna av parenteser

# Grammatik för aritmetiska uttryck, version 3

Ny, bättre, grammatik för aritmetiska uttryck:

$\text{Expr} \rightarrow \text{Term} \mid$

$\text{Expr} + \text{Term} \mid$

$\text{Expr} - \text{Term}$

$\text{Term} \rightarrow \text{Factor} \mid$

$\text{Term} * \text{Factor} \mid$

$\text{Term} / \text{Factor}$

$\text{Factor} \rightarrow \text{Num} \mid (\text{Expr})$

# Grammatik för aritmetiska uttryck, version 3

Ny, bättre, grammatik för aritmetiska uttryck:

$\text{Expr} \rightarrow \text{Term} \mid$

$\text{Expr} + \text{Term} \mid$

$\text{Expr} - \text{Term}$

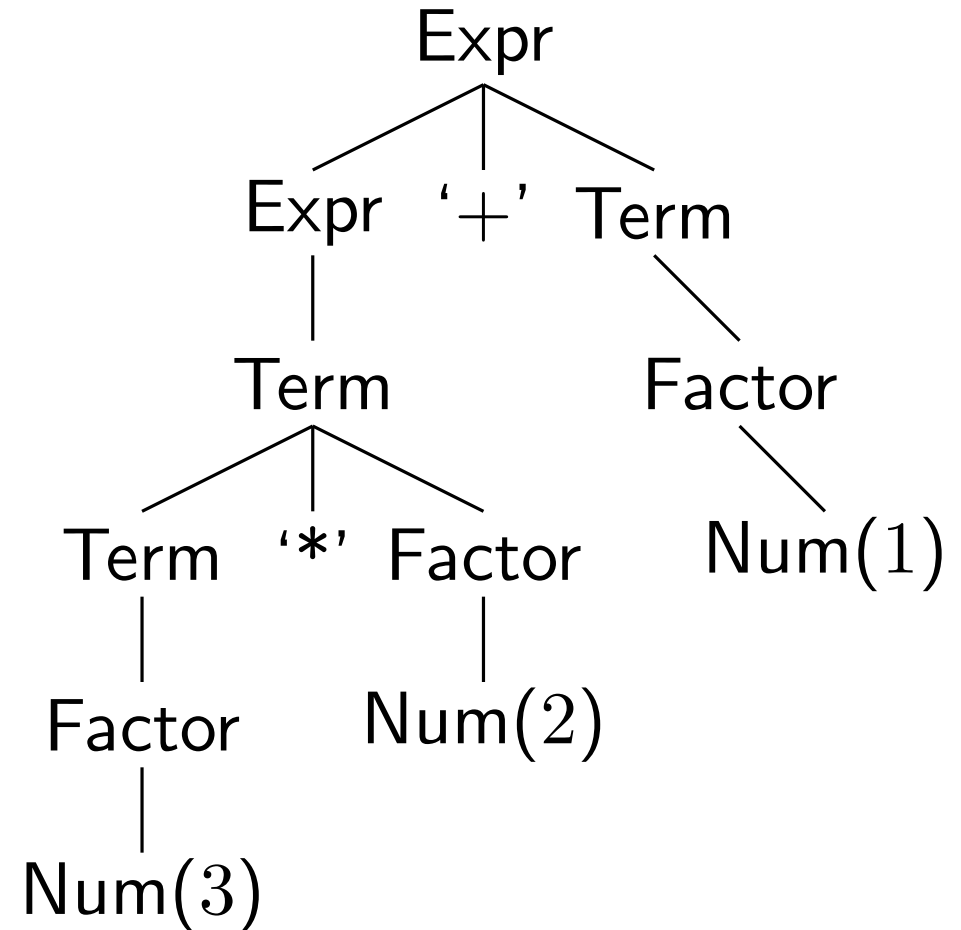
$\text{Term} \rightarrow \text{Factor} \mid$

$\text{Term} * \text{Factor} \mid$

$\text{Term} / \text{Factor}$

$\text{Factor} \rightarrow \text{Num} \mid (\text{Expr})$

Syntaxträd för  $3*2+1$ :



# Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

# Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

Har vi tur (eller otur...) och hittar två olika syntax-träd för samma sträng så vet vi att grammatiken är tvetydig

# Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

Har vi tur (eller otur...) och hittar två olika syntax-träd för samma sträng så vet vi att grammatiken är tvetydig

Men det finns inget enkelt sätt att övertyga sig om att en grammatik är otvetydig

(Att avgöra om en grammatik är tvetydig är ett så kallat *oavgörbart* problem – det existerar ingen algoritm för att göra detta.)

# Rekursiv medåkning för aritmetiska uttryck?

För att skriva en parser för detta med rekursiv medåkning ska vi ha tre funktioner `Expr`, `Term`, `Factor`

```
Expr → Term |  
      Expr + Term |  
      Expr - Term  
Term  → Factor |  
      Term * Factor |  
      Term / Factor  
Factor → Num | (Expr)
```

# Rekursiv medåkning för aritmetiska uttryck?

För att skriva en parser för detta med rekursiv medåkning ska vi ha tre funktioner `Expr`, `Term`, `Factor`

```
Expr → Term |  
      Expr + Term |  
      Expr - Term  
Term → Factor |  
      Term * Factor |  
      Term / Factor  
Factor → Num | (Expr)
```

```
ParseTree Expr() {  
    Token t = NextToken();  
    // TODO: hur veta vilken produktionsregel  
    //       som ska användas???  
}
```



# Begränsningar för rekursiv medåkning

**Problem 1** för rekursiv medåkning:  
*Vänster-rekursion*: ett Expr kan börja med ett Expr (och samma för Term)

Expr  $\rightarrow$  Term |

Expr + Term |

Expr - Term

Term  $\rightarrow$  Factor |

Term \* Factor |

Term / Factor

Factor  $\rightarrow$  Num | (Expr)

# Begränsningar för rekursiv medåkning

**Problem 1** för rekursiv medåkning:  
*Vänster-rekursion*: ett Expr kan börja med ett Expr (och samma för Term)

**Problem 2** för rekursiv medåkning:  
Två produktionsregler som kan börja på samma tecken, t.ex.

```
<Thing> ::= "a" <Something>
          | "a" <Another>
          | "b" <Thing>
```

När vi läser ett "a" kan vi inte veta vilken av de två första reglerna som ska användas.

```
Expr → Term |
      Expr + Term |
      Expr - Term
Term → Factor |
      Term * Factor |
      Term / Factor
Factor → Num | (Expr)
```

# Begränsningar för rekursiv medåkning

**Problem 1** för rekursiv medåkning:  
*Vänster-rekursion*: ett Expr kan börja med ett Expr (och samma för Term)

**Problem 2** för rekursiv medåkning:  
Två produktionsregler som kan börja på samma tecken, t.ex.

```
<Thing> ::= "a" <Something>
          | "a" <Another>
          | "b" <Thing>
```

När vi läser ett "a" kan vi inte veta vilken av de två första reglerna som ska användas.

Grammatiken kan inte användas med rekursiv medåkning, vi måste skriva om den (igen...)

```
Expr → Term |
      Expr + Term |
      Expr - Term
Term → Factor |
      Term * Factor |
      Term / Factor
Factor → Num | (Expr)
```

# EBNF

Problemet är främst vårt sätt att beskriva listor (en Expr är en lista av Termer, separerade av +/-)

# EBNF

Problemet är främst vårt sätt att beskriva listor (en Expr är en lista av Termer, separerade av +/-)

En behändig notation för grammatiker där det är lätt att beskriva listor: EBNF (Utökad Backus-Naur-Form)

# EBNF

Problemet är främst vårt sätt att beskriva listor (en Expr är en lista av Termer, separerade av +/-)

En behändig notation för grammatiker där det är lätt att beskriva listor: EBNF (Utökad Backus-Naur-Form)

Har några praktiska tillägg till BNF. Av intresse för oss just nu:

```
<List> ::= <Elem> { ", " <Elem> }
```

# EBNF

Problemet är främst vårt sätt att beskriva listor (en Expr är en lista av Termer, separerade av +/-)

En behändig notation för grammatiker där det är lätt att beskriva listor: EBNF (Utökad Backus-Naur-Form)

Har några praktiska tillägg till BNF. Av intresse för oss just nu:

`<List> ::= <Elem> { ", " <Elem > }`

Notationen {ole dole doff} betyder  
“ole dole doff upprepat 0 eller fler gånger”

# EBNF

Problemet är främst vårt sätt att beskriva listor (en Expr är en lista av Termer, separerade av +/-)

En behändig notation för grammatiker där det är lätt att beskriva listor: EBNF (Utökad Backus-Naur-Form)

Har några praktiska tillägg till BNF. Av intresse för oss just nu:

```
<List> ::= <Elem> { "," <Elem> }
```

Betyder alltså:

```
<List> ::= <Elem> | <Elem> "," <Elem> | <Elem> "," Elem "," Elem | E
```



# Grammatik för aritmetiska uttryck, försök 4

```
<Expr> ::= <Term> { <PM> <Term> }  
<Term> ::= <Factor> { <MD> <Factor> }  
<Factor> ::= NUM | "(" <Expr> ")"  
<PM> ::= "+" | "-"  
<MD> ::= "*" | "/"
```

Java-kod för att parse ett Expr:

```
ParseTree Expr() {  
    ParseTree result = Term();  
    while (peekToken() == PLUS or  
           peekToken() == MINUS) {  
        Token operator = nextToken();  
        ParseTree next = Term();  
        result = new BinaryOperation(operator, result, next);  
    }  
    return result;  
}
```

# Rekursiv medåkning för aritmetiska uttryck

```
ParseTree Expr() {
    ParseTree result = Term();
    while (peekToken() == PLUS or
           peekToken() == MINUS) {
        Token op = nextToken();
        ParseTree next = Term();
        result = new BinOp(op,
                           result,
                           next);
    }
    return result;
}
```

```
ParseTree Term() {
    ParseTree result = Factor();
    while (peekToken() == TIMES or
           peekToken() == DIVIDE) {
        Token op = nextToken();
        ParseTree next = Factor();
        result = new BinOp(op,
                           result,
                           next);
    }
    return result;
}
```

```
ParseTree Factor() {
    Token t = nextToken();
    if (t == NUM) return new Number(t.value);
    else if (t == LPAREN) {
        ParseTree result = Expr();
        if (nextToken() != RPAREN)
            throw new SyntaxError();
        return result;
    }
    throw new SyntaxError();
}
```

# Rekursiv medåkning för aritmetiska uttryck

```
ParseTree Expr() {
    ParseTree result = Term();
    while (peekToken() == PLUS or
           peekToken() == MINUS) {
        Token op = nextToken();
        ParseTree next = Term();
        result = new BinOp(op,
                           result,
                           next);
    }
    return result;
}
```

```
ParseTree Term() {
    ParseTree result = Factor();
    while (peekToken() == TIMES or
           peekToken() == DIVIDE) {
        Token op = nextToken();
        ParseTree next = Factor();
        result = new BinOp(op,
                           result,
                           next);
    }
    return result;
}
```

```
ParseTree Factor() {
    Token t = nextToken();
    if (t == NUM) return new Number(t.value);
    else if (t == LPAREN) {
        ParseTree result = Expr();
        if (nextToken() != RPAREN)
            throw new SyntaxError();
        return result;
    }
    throw new SyntaxError();
}
```

Fullständig Java-implementation upplagd på kurshemsidan. Provkör med några olika uttryck och försök få en känsla för vad som händer!!

# Nästa föreläsning

Lite mer teori

- Stackautomater
- Olika språk-klasser

Kanske lite om parser-generatorer

Sammanfattning av kursavsnittet och anvisningar för kontrollskrivningen