Vi kommer jobba med filen rdparse.rb från kurssidan under föreläsningen

TDP007 Konstruktion av datorspråk
Föreläsning 6

LINKÖPINGS UNIVERSITET

Parsning

TDP007 Konstruktion av datorspråk
Föreläsning 6

LINKÖPINGS UNIVERSITET

Pontus Haglund och Love Arreborn
Institutionen för datavetenskap
2025-02-06

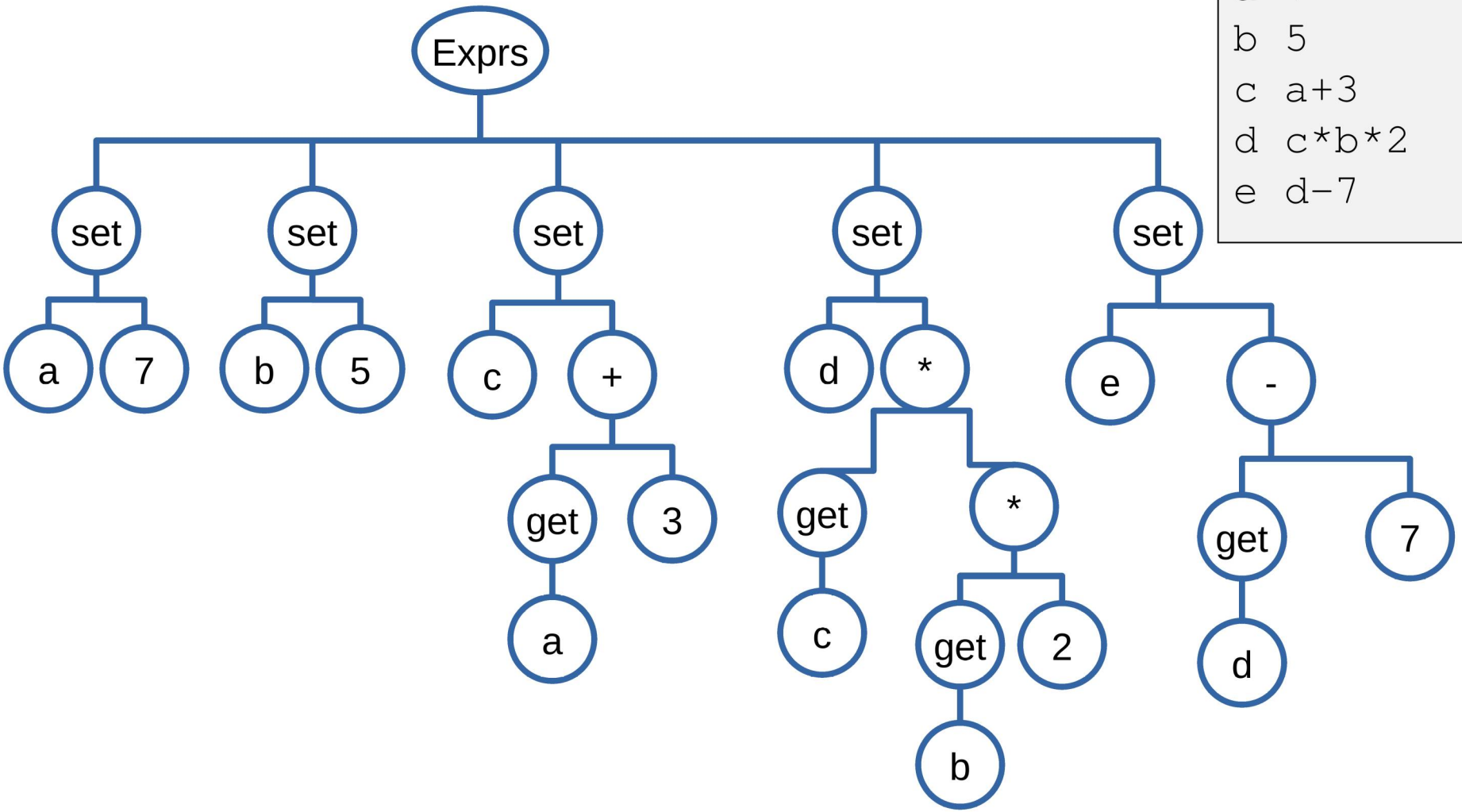
Om eval i olika kontexter

- **eval( )**
  - kör koden i den globala kontexten, ungefär som om man skrivit in den vid prompten i irb *(definierad i klassen Kernel)*
- ***o*.instance_eval( )**
  - kör koden i kontexten av objektet *o*, vilket innebär att man t.ex. kan komma åt instansvariabler *(definierad i klassen BasicObject)*
- ***C*.class_eval( )**
  - kör koden i kontexten av klassen *C*, vilket innebär att man t.ex. kan komma åt klassvariabler och definiera nya metoder *(definierad i klassen Module)*

Det finns även en module_eval som körs i kontexten av en modul. Några av dessa metoder kan förutom en sträng även ta ett block.

Hela programmet som ett träd (läs)

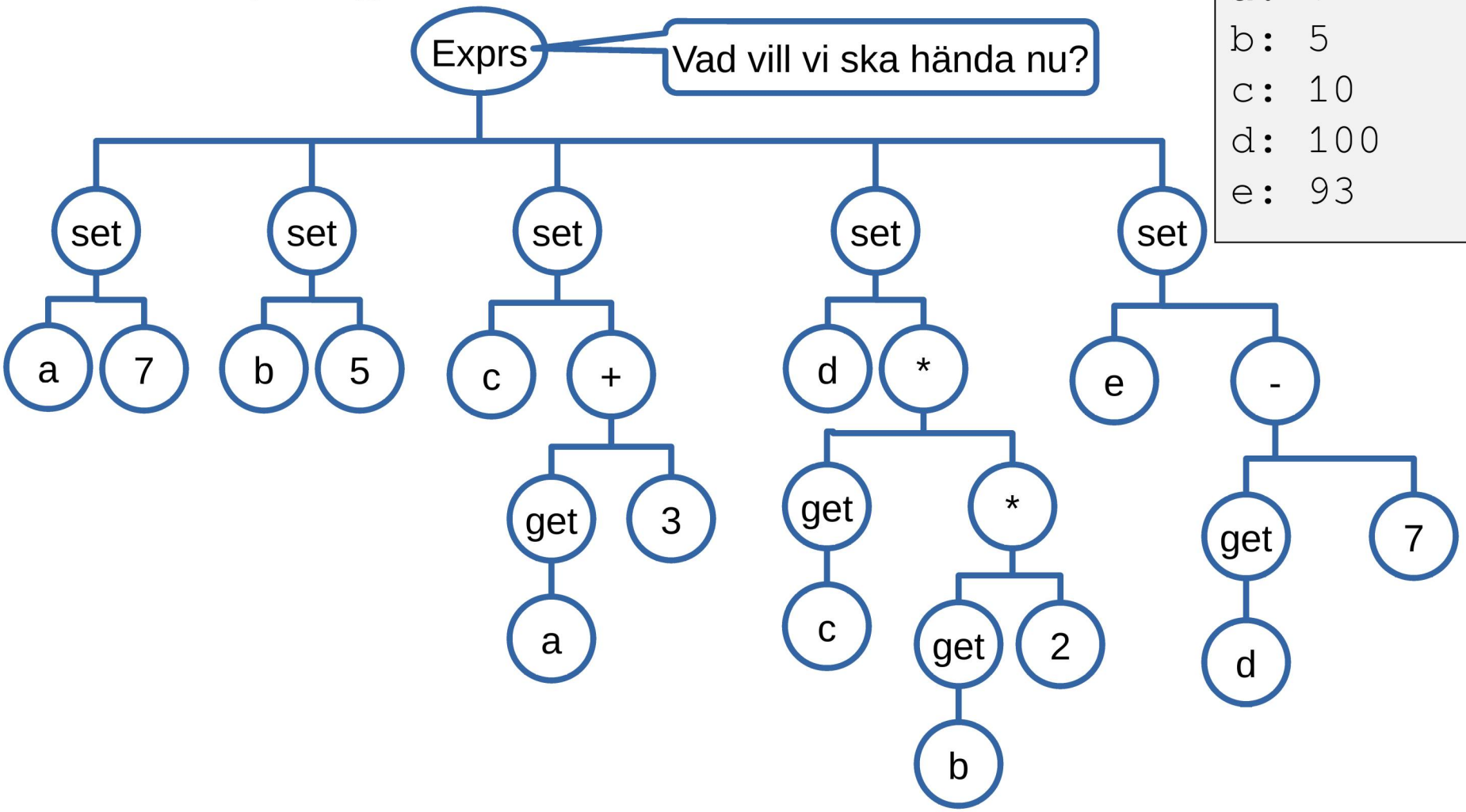
```
a 7
b 5
c a+3
d c*b*2
e d-7
```



Kör programmet

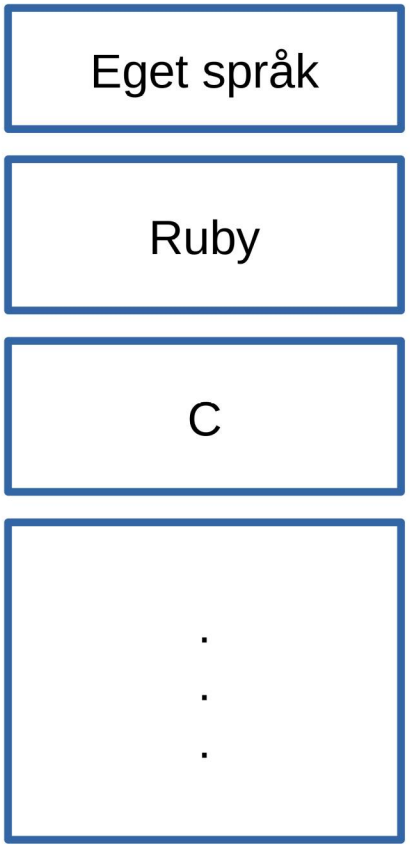
DATA:

```
a: 7
b: 5
c: 10
d: 100
e: 93
```



Dagens föreläsning och seminarie 3

- Stark relation till TDP019
- Skapa egna språk ovanpå rubys interpretator
- Även seminarie 4 kommer inkludera utbyggnad av interpretatorn



Analys av källkod

- Lexikalisk analys
  - *Bildar tokens utifrån källkodens text.*
- Syntaktisk analys *(parsning)*
  - *Kontrollerar att koden är syntaktiskt korrekt*
  - *Bygger upp en datastruktur (oftast en trädstruktur) som motsvarar källkoden för vidare bearbetning i nästa steg.*
- Semantisk analys
  - *Kontrollerar t.ex. typfel.*
- Fler steg, beroende på uppgiften…

Ett exempel

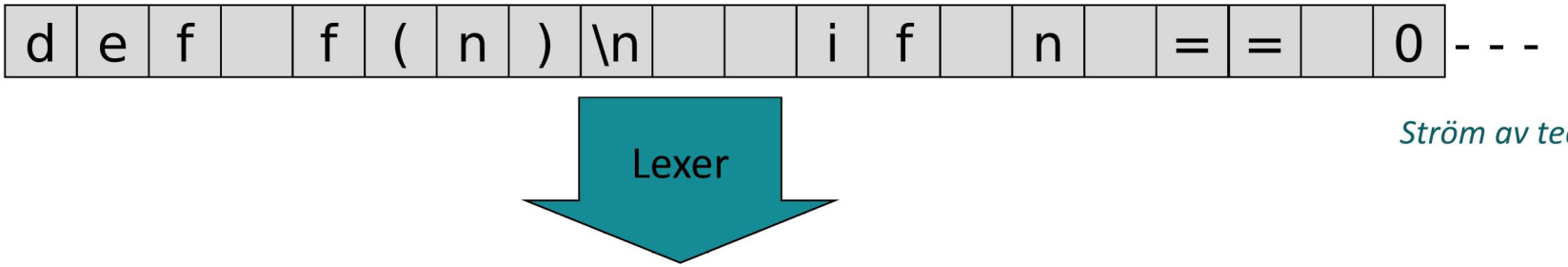
Föreställ dig att du har följande kod i en fil:

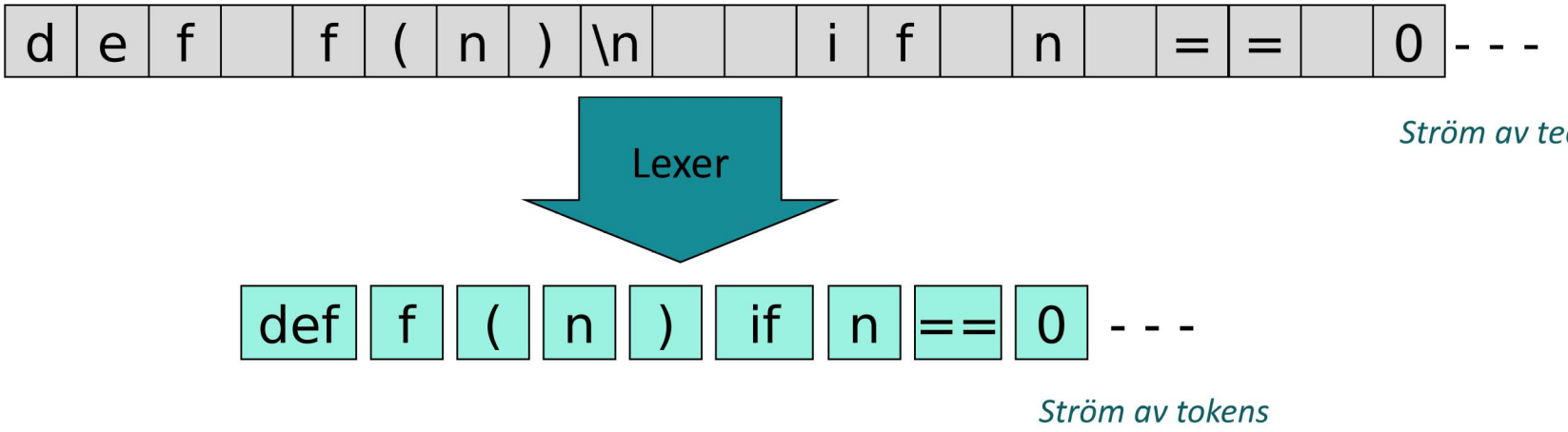
```
def f(n)
  if n == 0 then
   puts ”hej”
  end
end
```

Hur ska en dator läsa och förstå denna fil?

d	e	f		f	(	n	)	\n			i	f		n		=	=		0	- - -

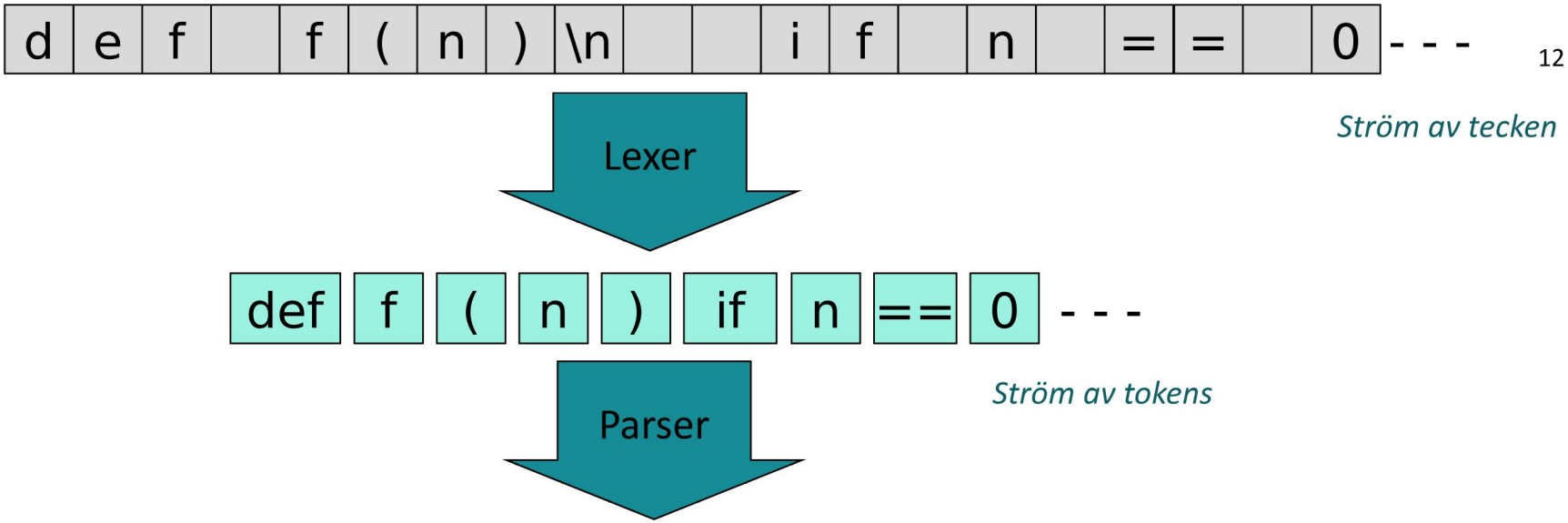
Ström av tecken

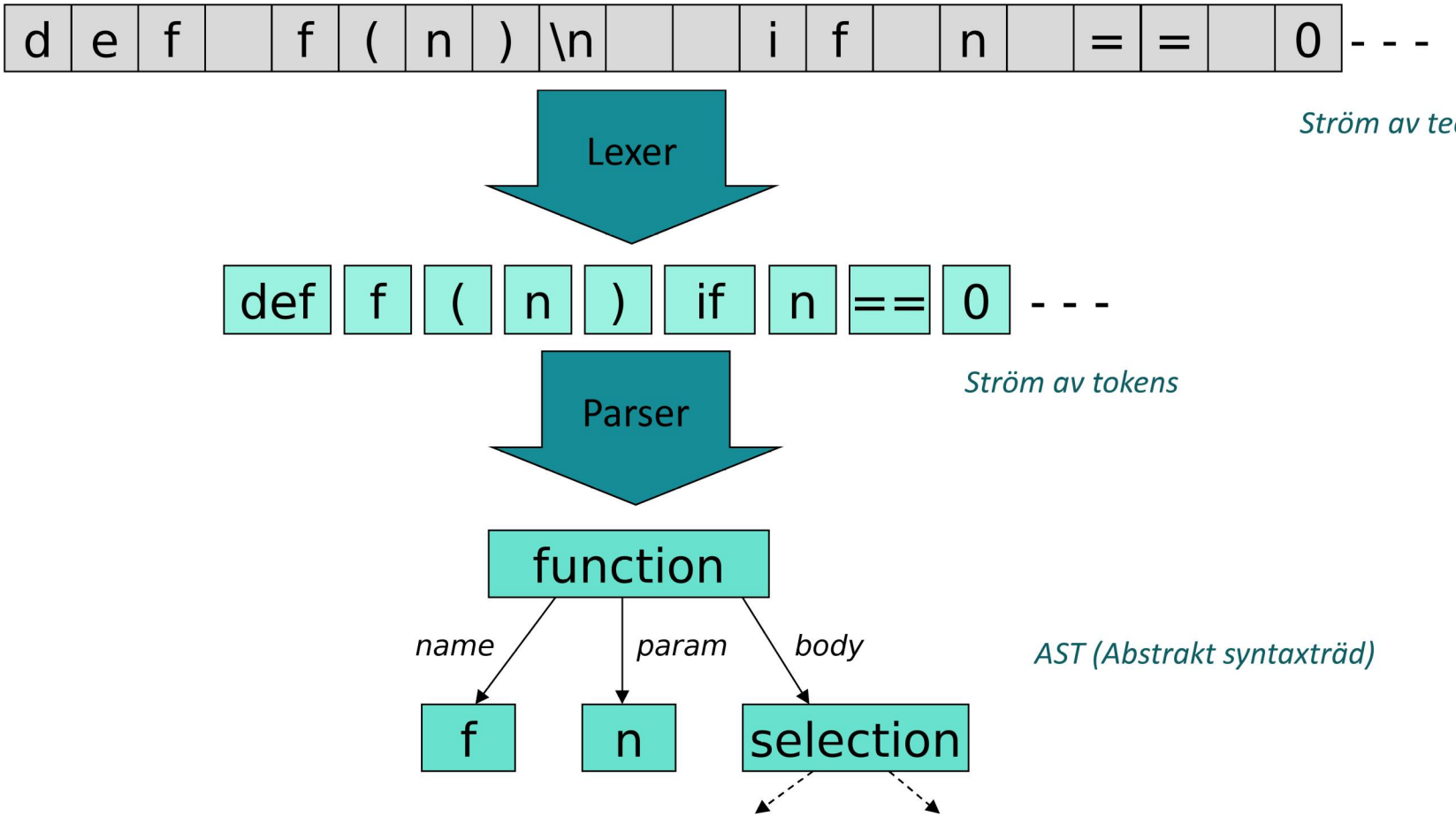

d e f f (n) \n i f n = = 0 - - -
Lexer
Ström av tecken

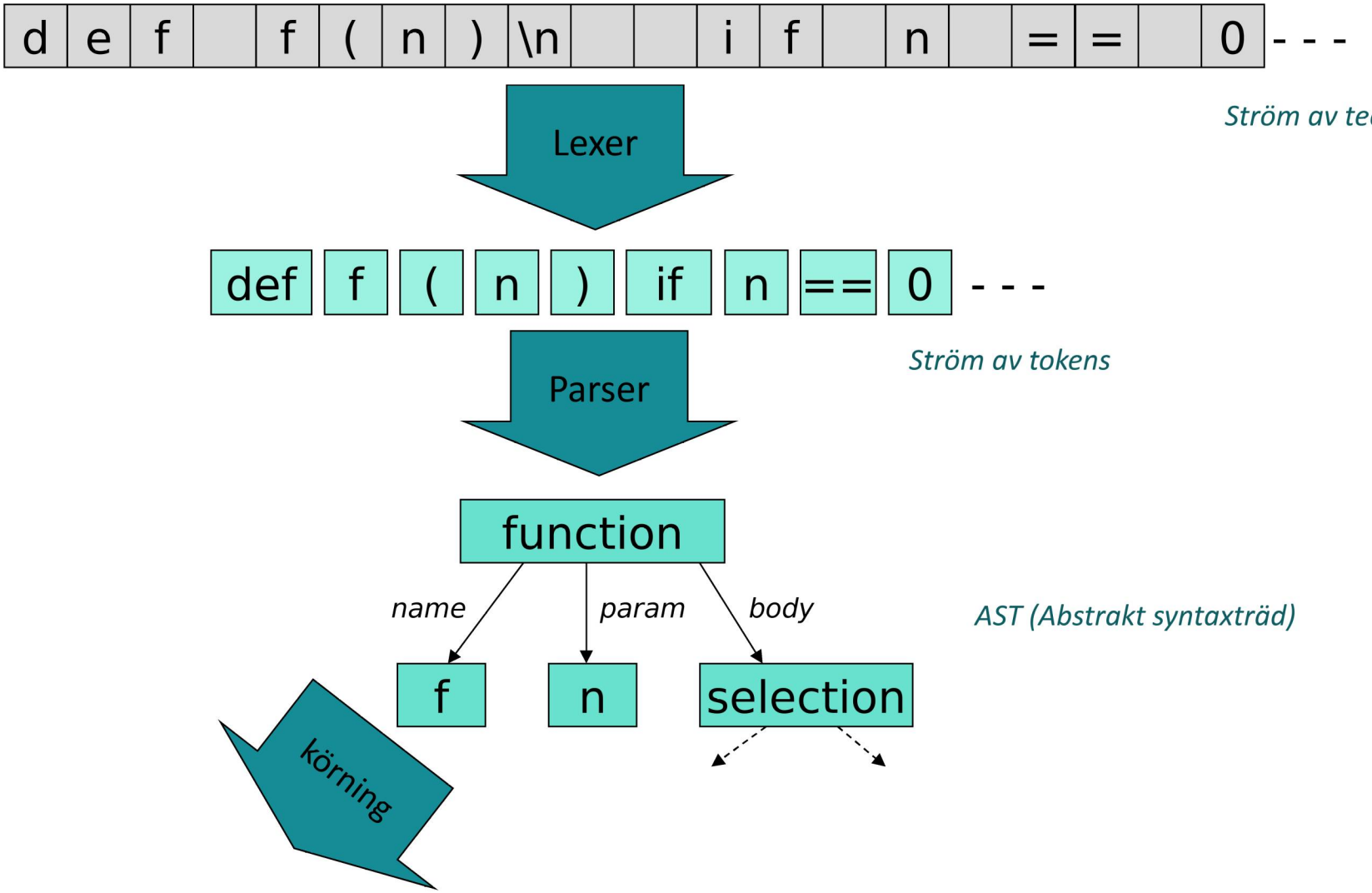


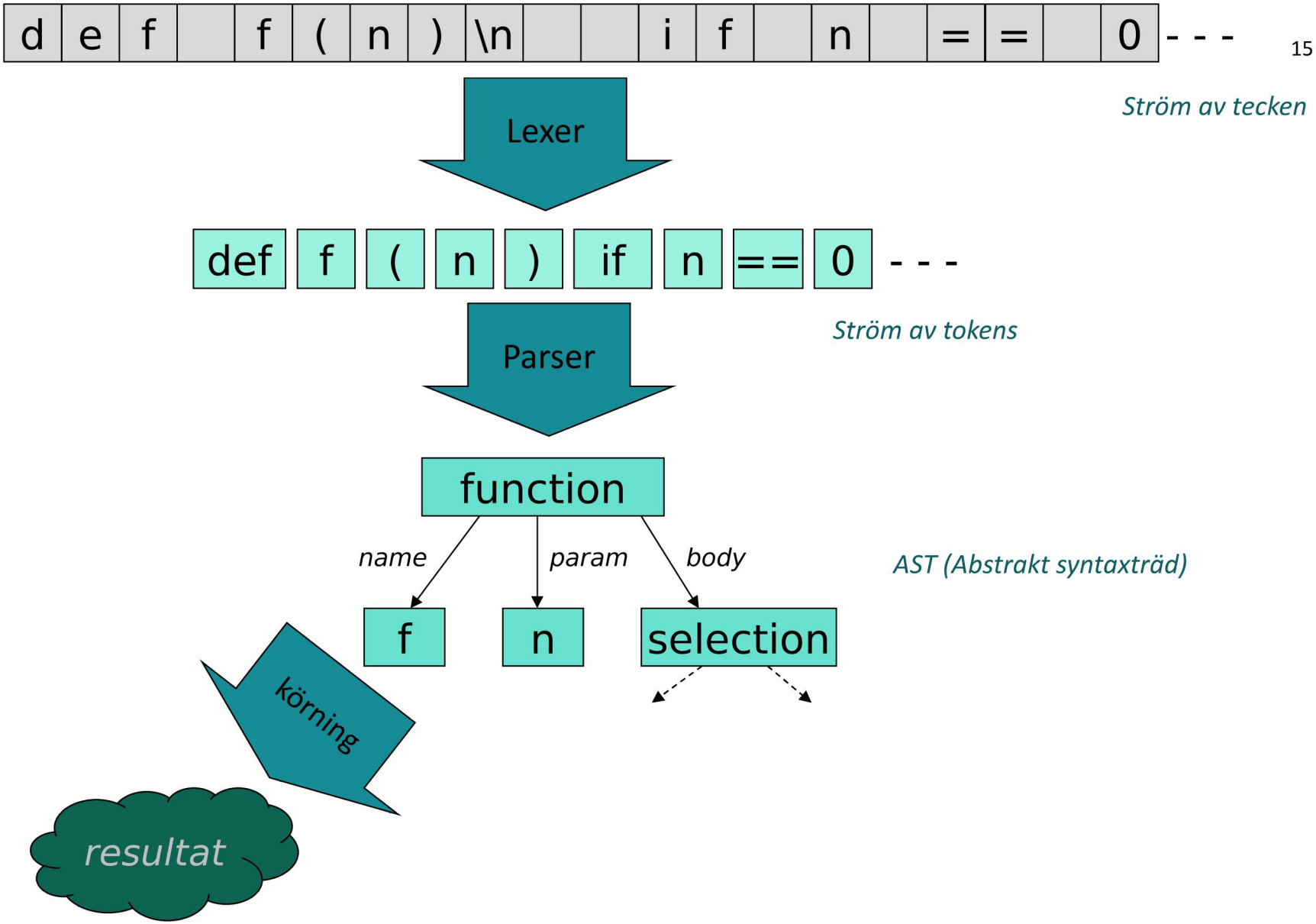
Ström av tecken

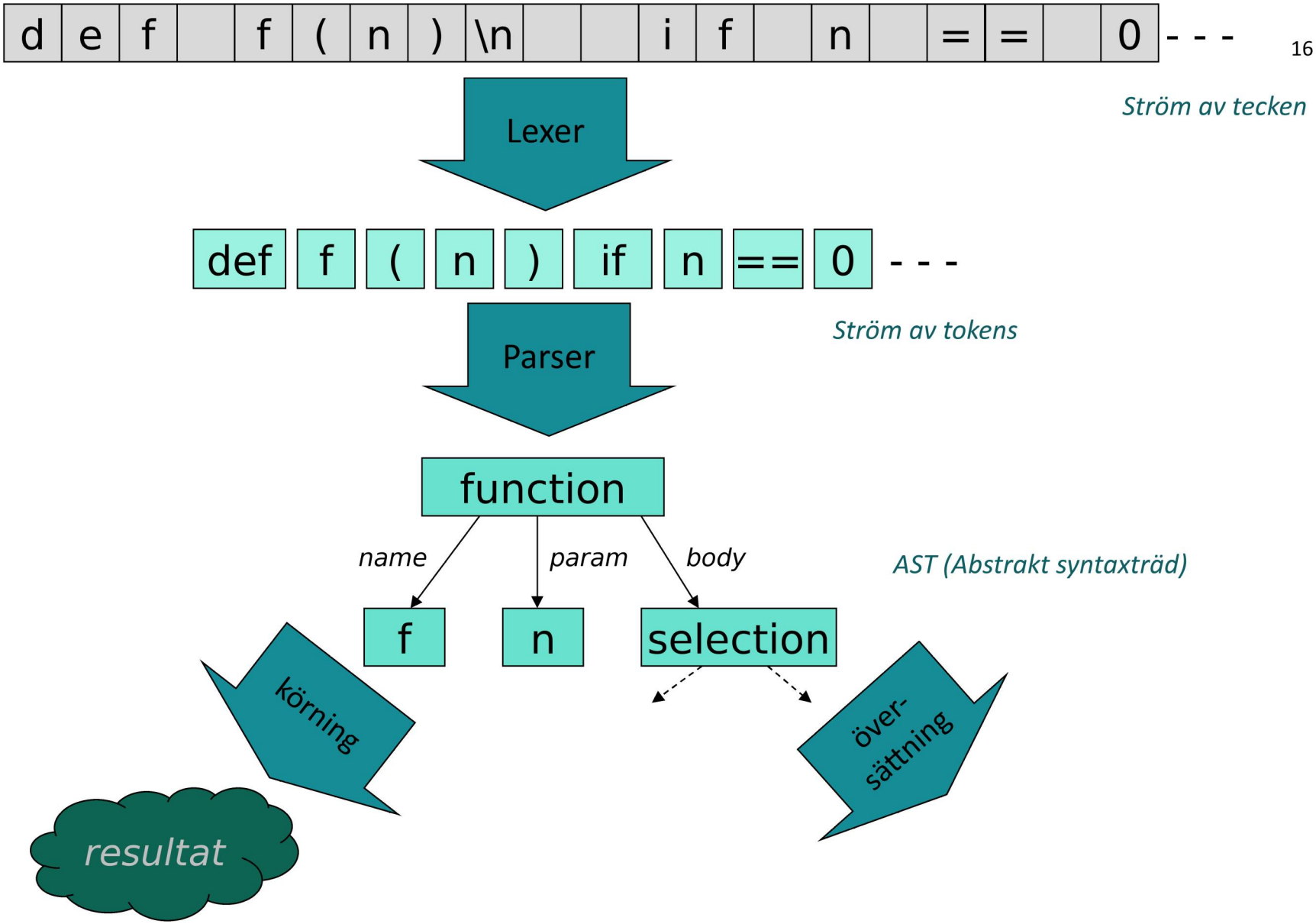
Ström av tokens

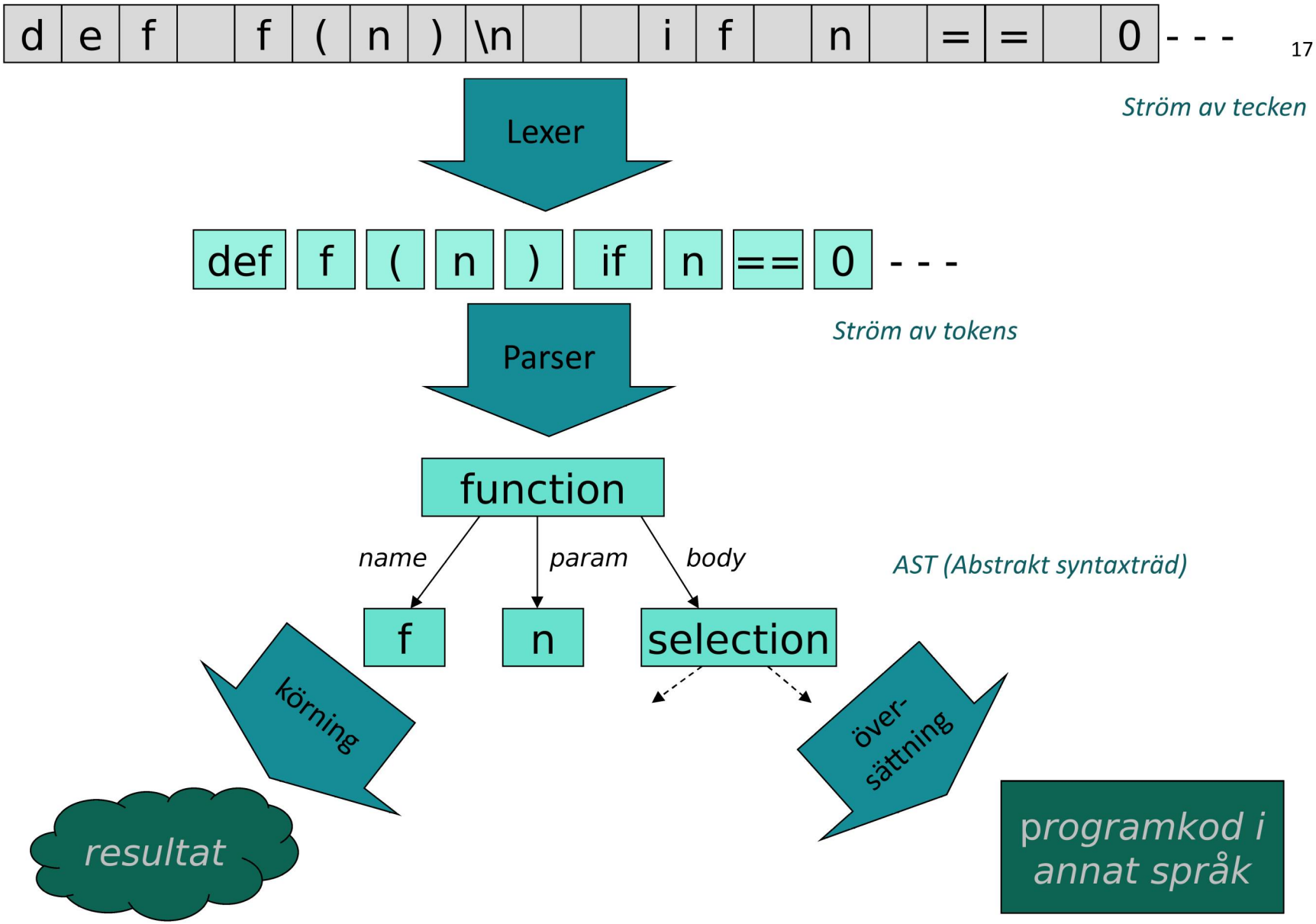

d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser

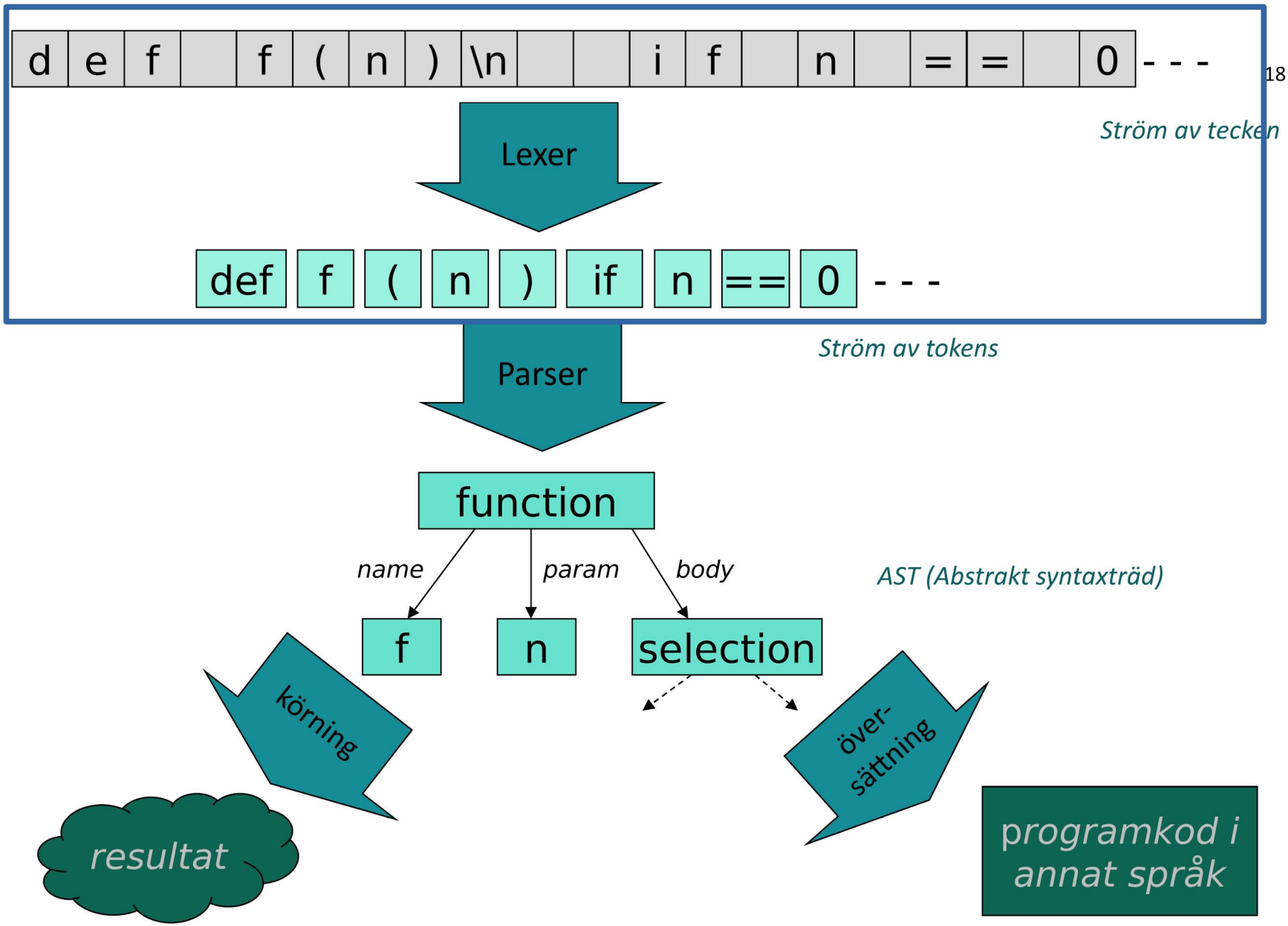

d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)


d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)
körning


d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)
körning
resultat


d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
AST (Abstrakt syntaxträd)
f
n
selection
körning
över-
sättning
resultat

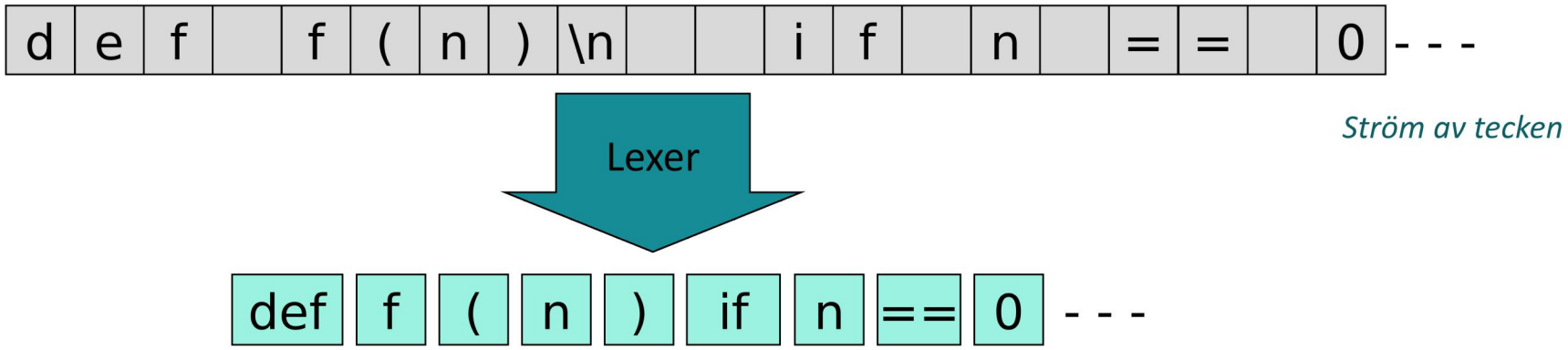

d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)
körning
över-
sättning
resultat
programkod i
annat språk


d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)
körning
över-
sättning
resultat
programkod i
annat språk

Lexer – Hur görs valen av tokens?

Behöver någon typ av specifikation som definerar hur en ström bryts ner

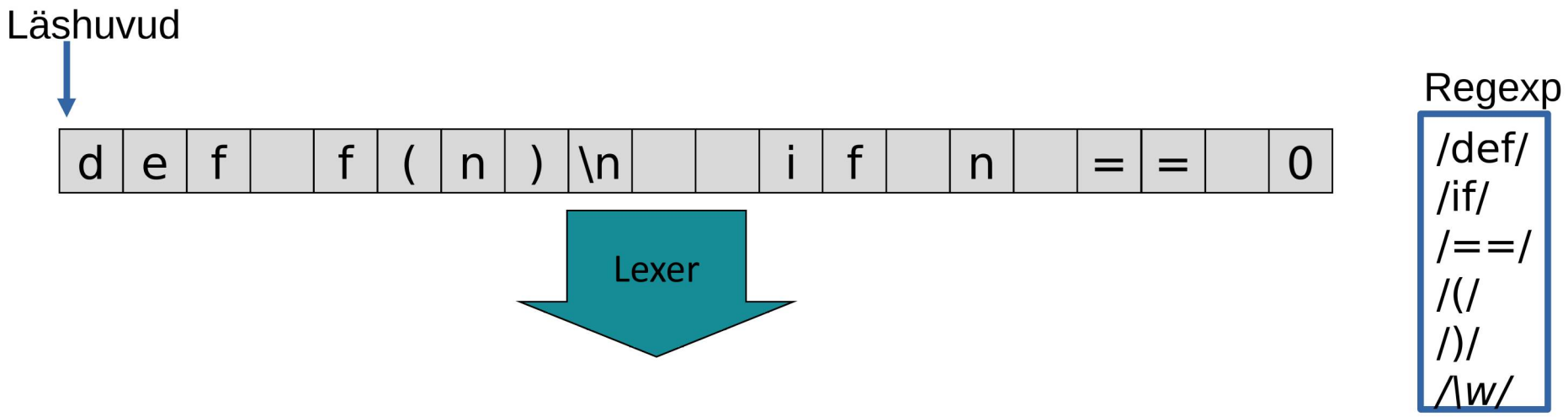
Ofta Regexp



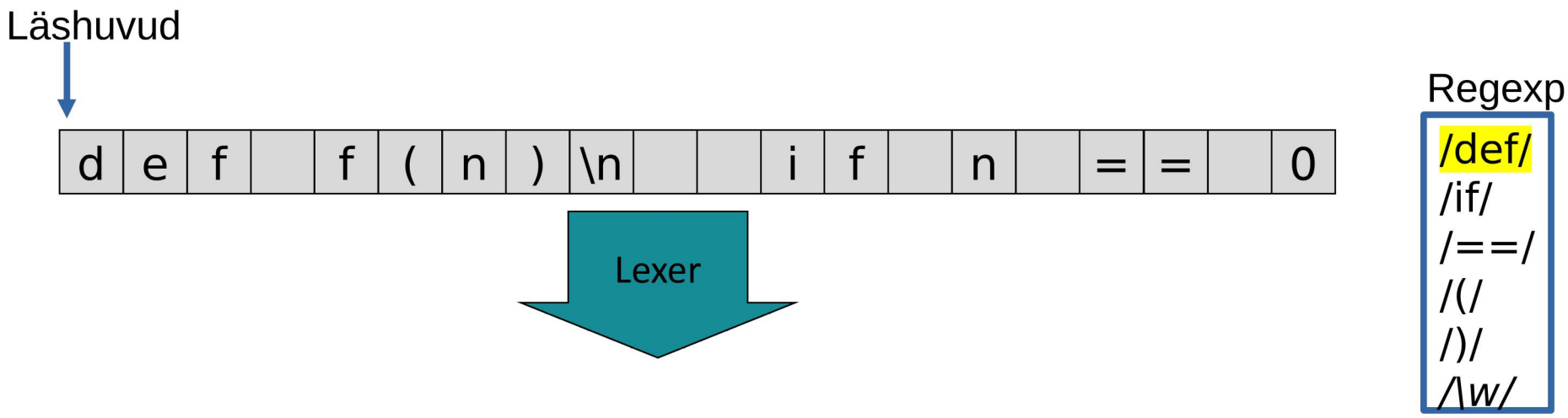
Ström av tecken

Ström av tokens

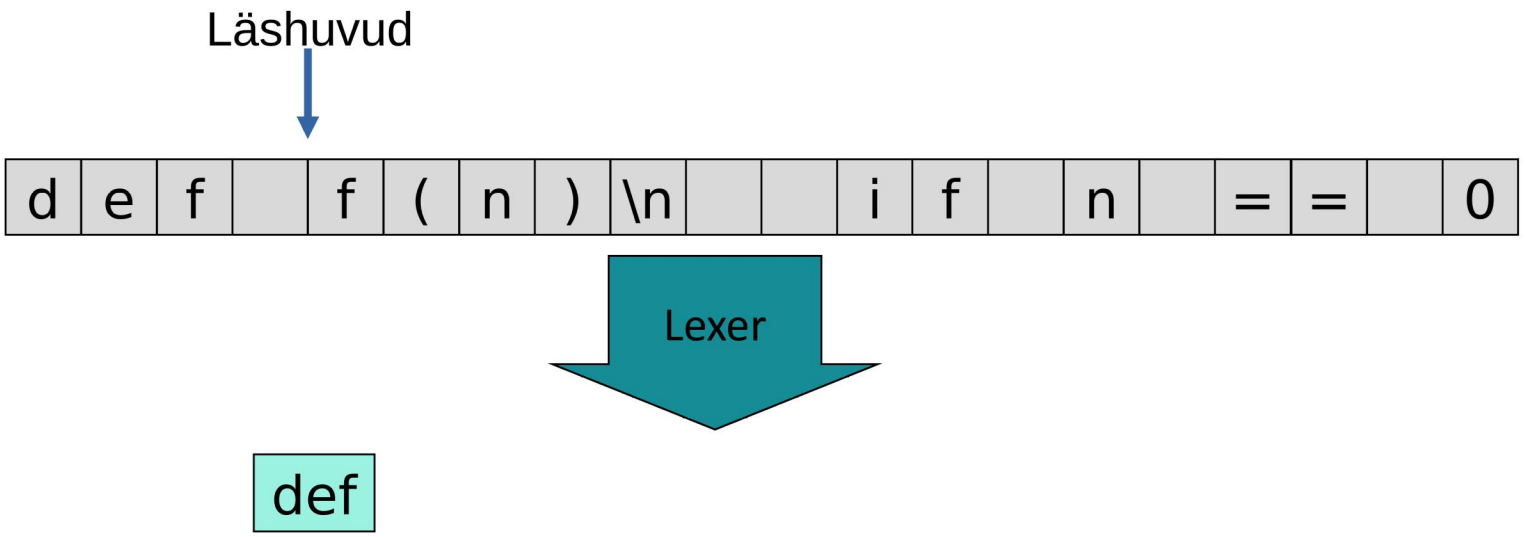
Lexer – Hur görs valen av tokens? (Regexp)



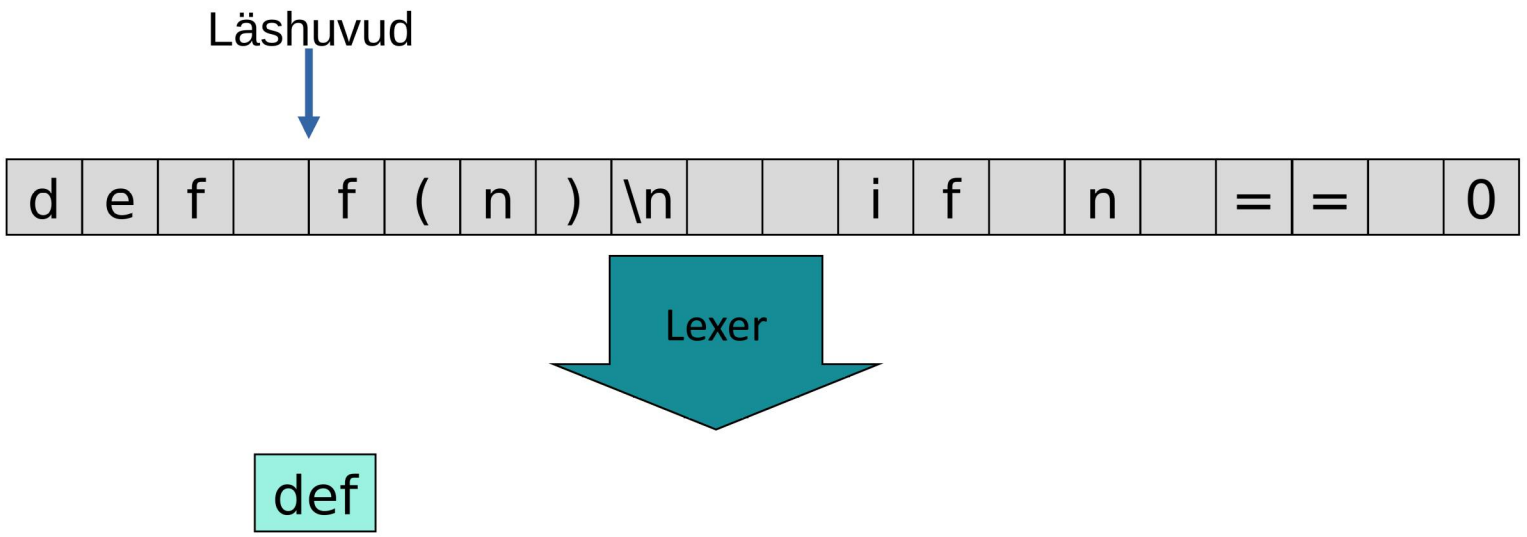
Lexer – Hur görs valen av tokens? (Regexp)



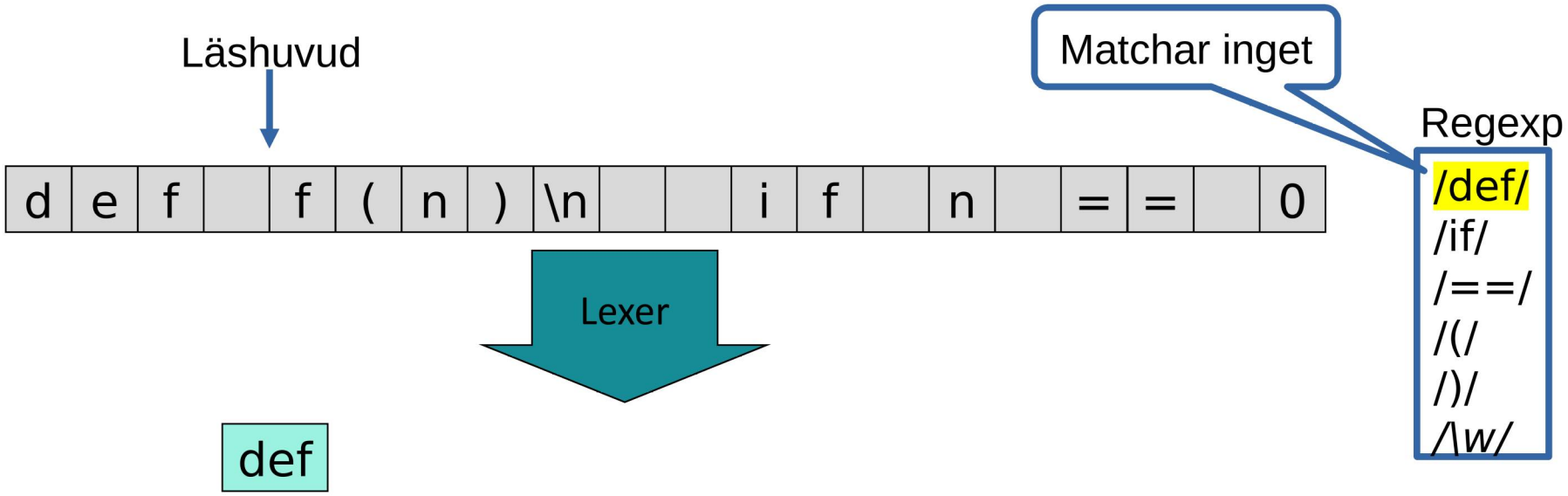
Lexer – Hur görs valen av tokens? (Regexp)



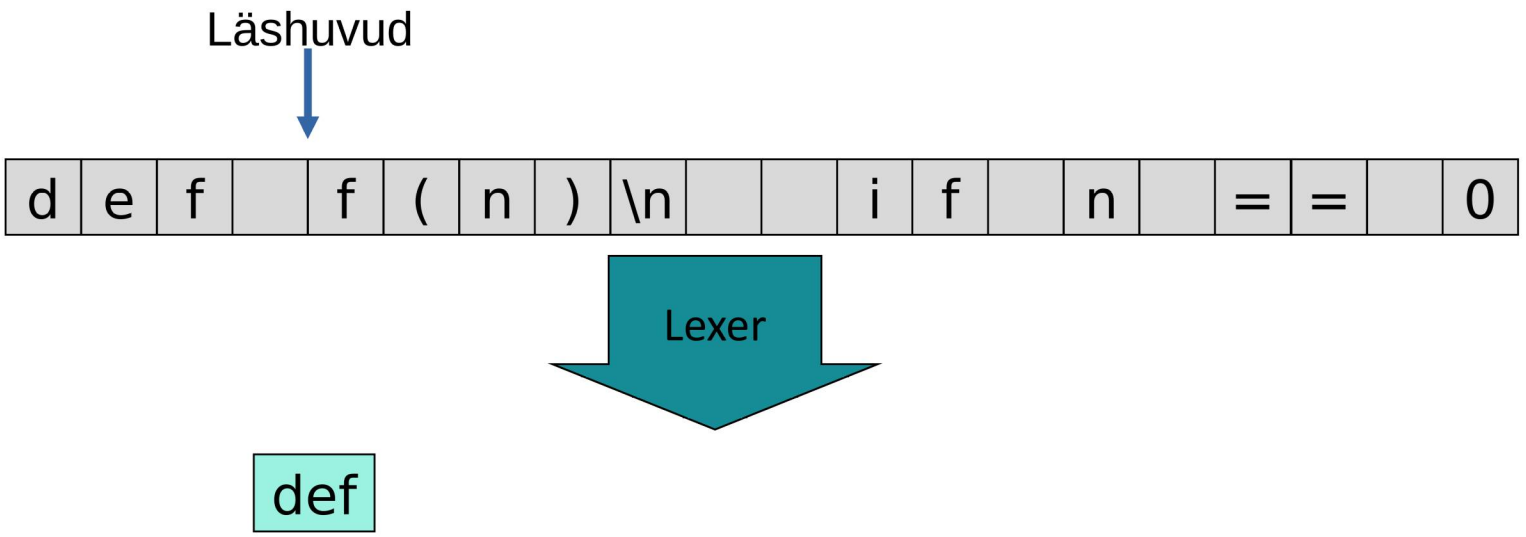
Lexer – Hur görs valen av tokens? (Regexp)



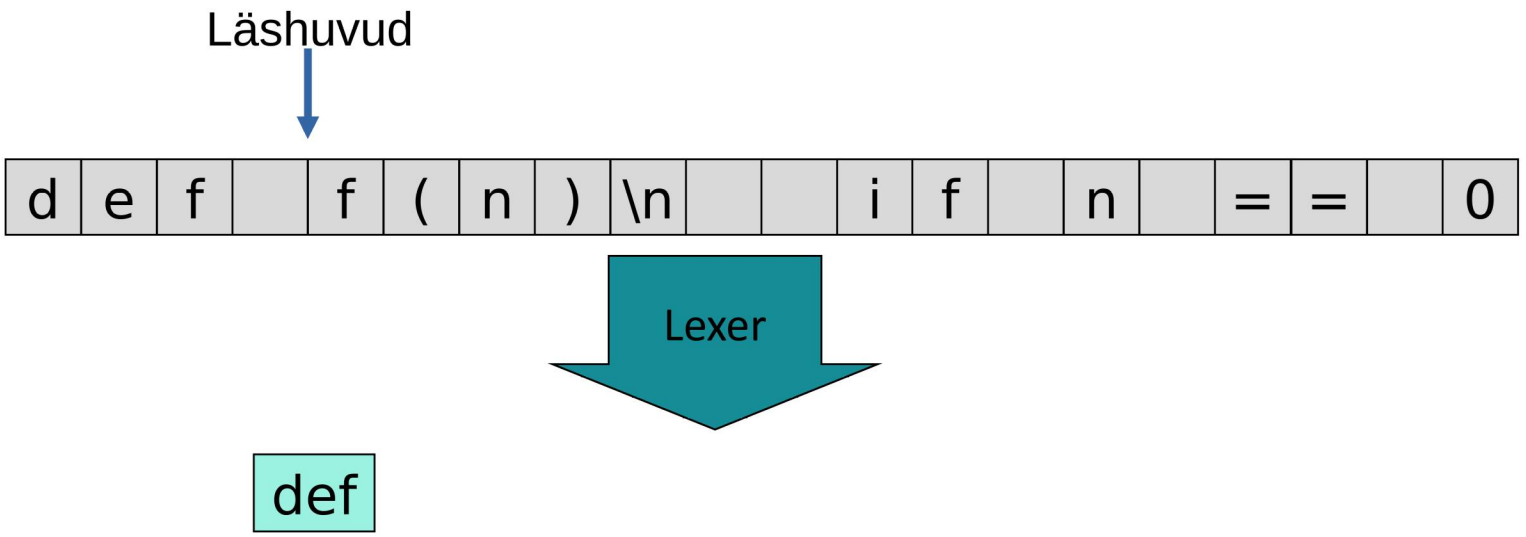
Lexer – Hur görs valen av tokens? (Regexp)



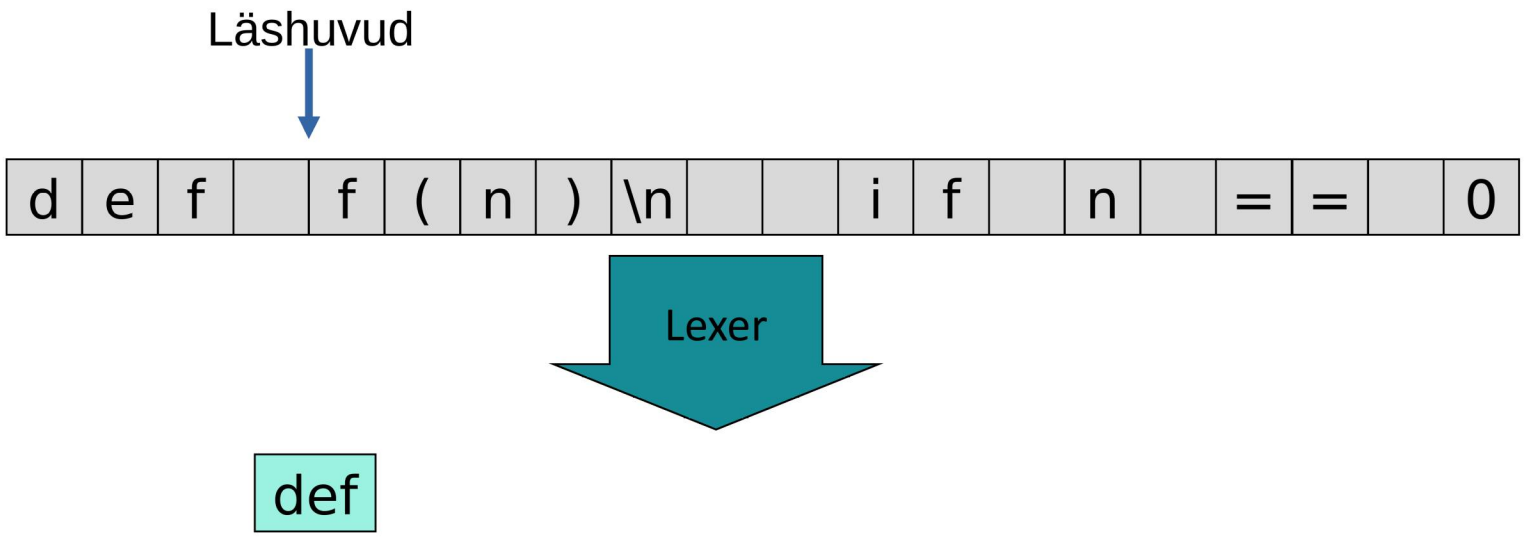
Lexer – Hur görs valen av tokens? (Regexp)



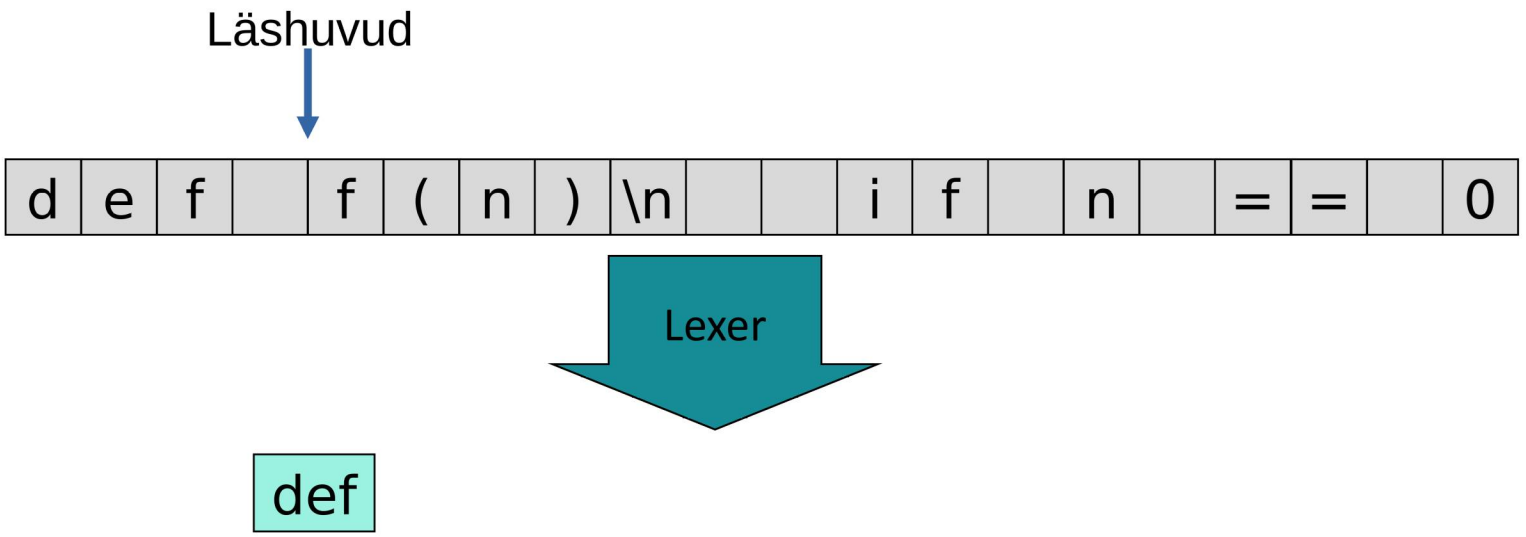
Lexer – Hur görs valen av tokens? (Regexp)



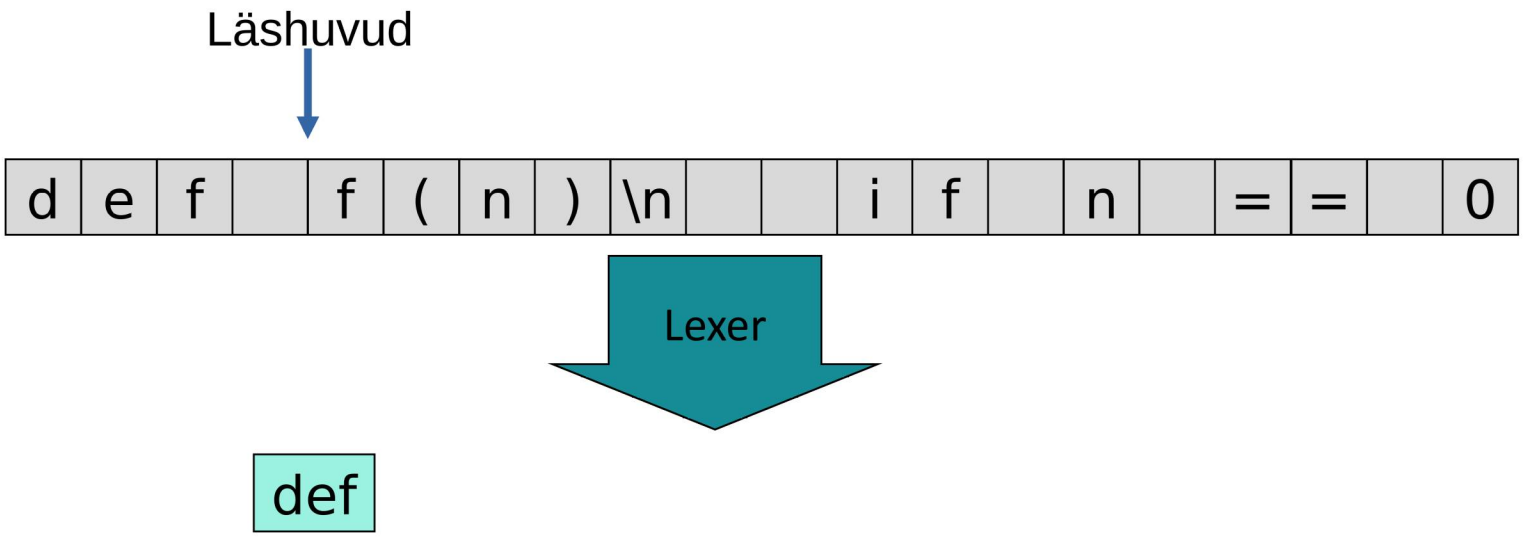
Lexer – Hur görs valen av tokens? (Regexp)



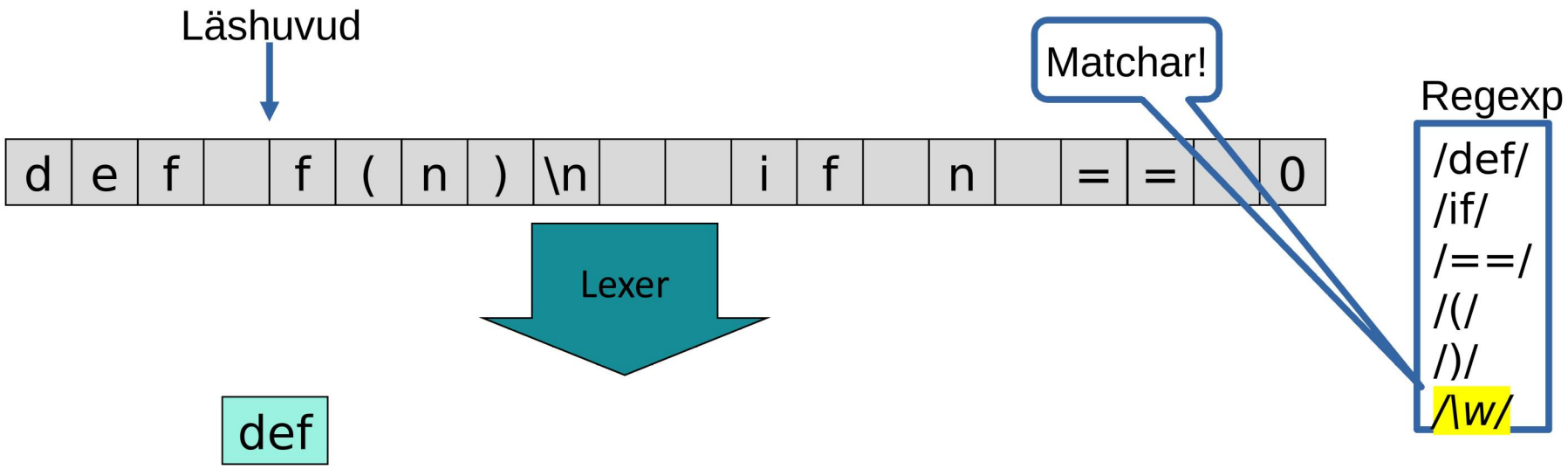
Lexer – Hur görs valen av tokens? (Regexp)



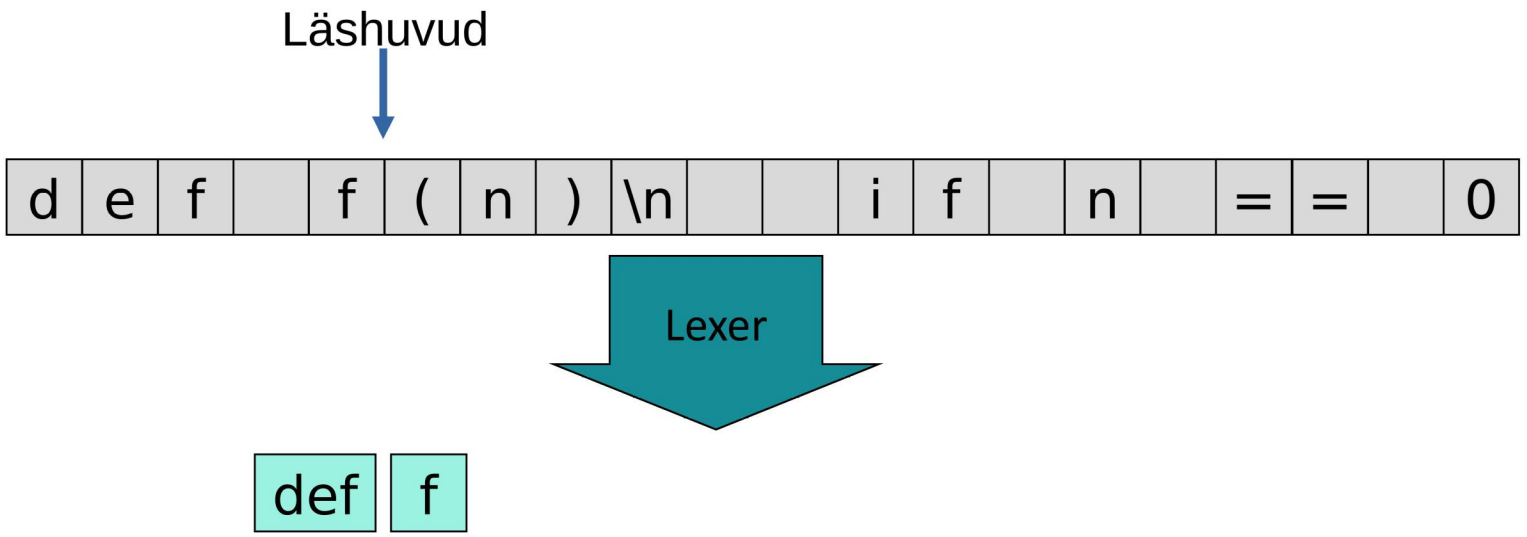
Lexer – Hur görs valen av tokens? (Regexp)



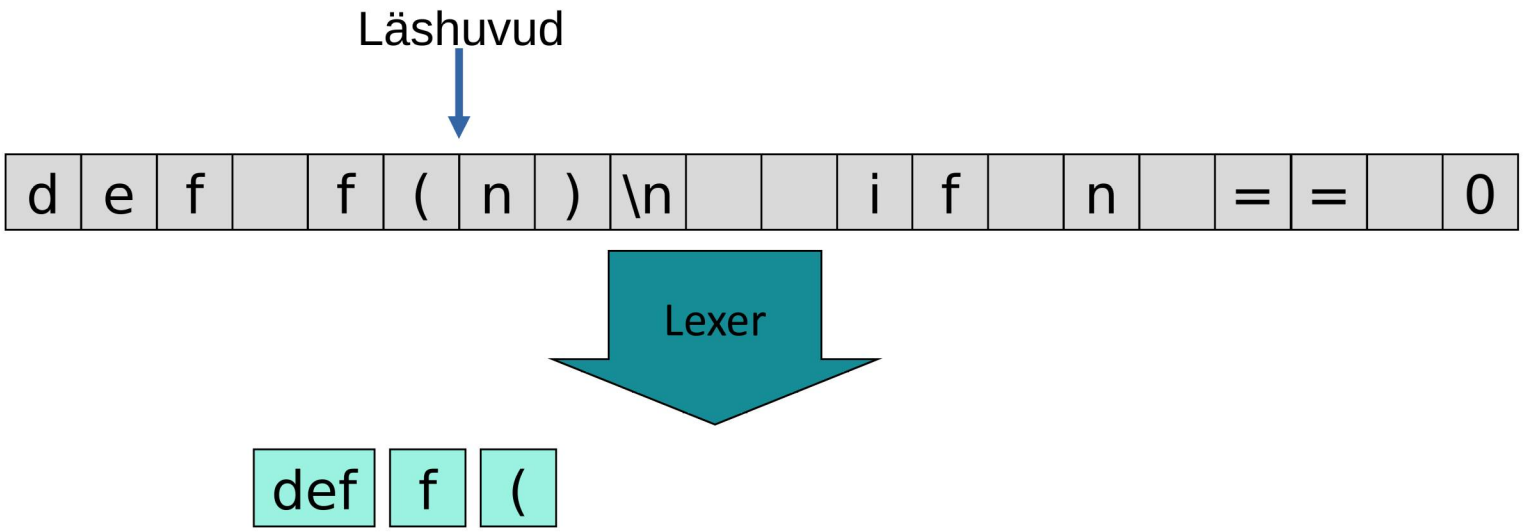
Lexer – Hur görs valen av tokens? (Regexp)



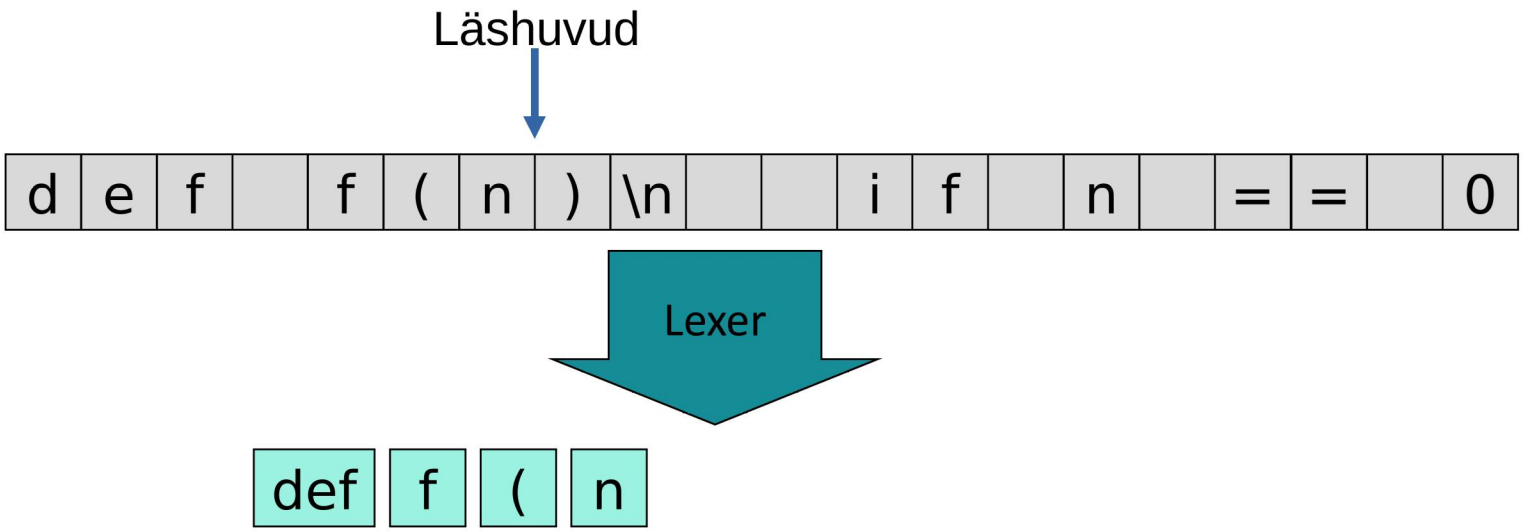
Lexer – Hur görs valen av tokens? (Regexp)



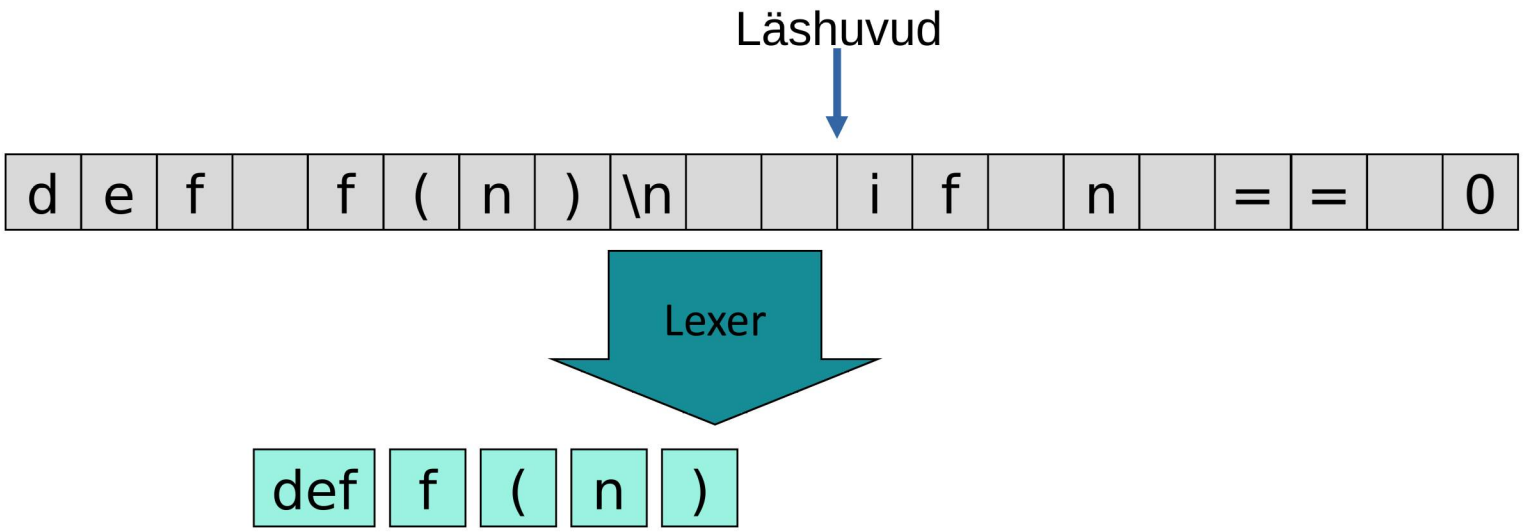
Lexer – Hur görs valen av tokens? (Regexp)



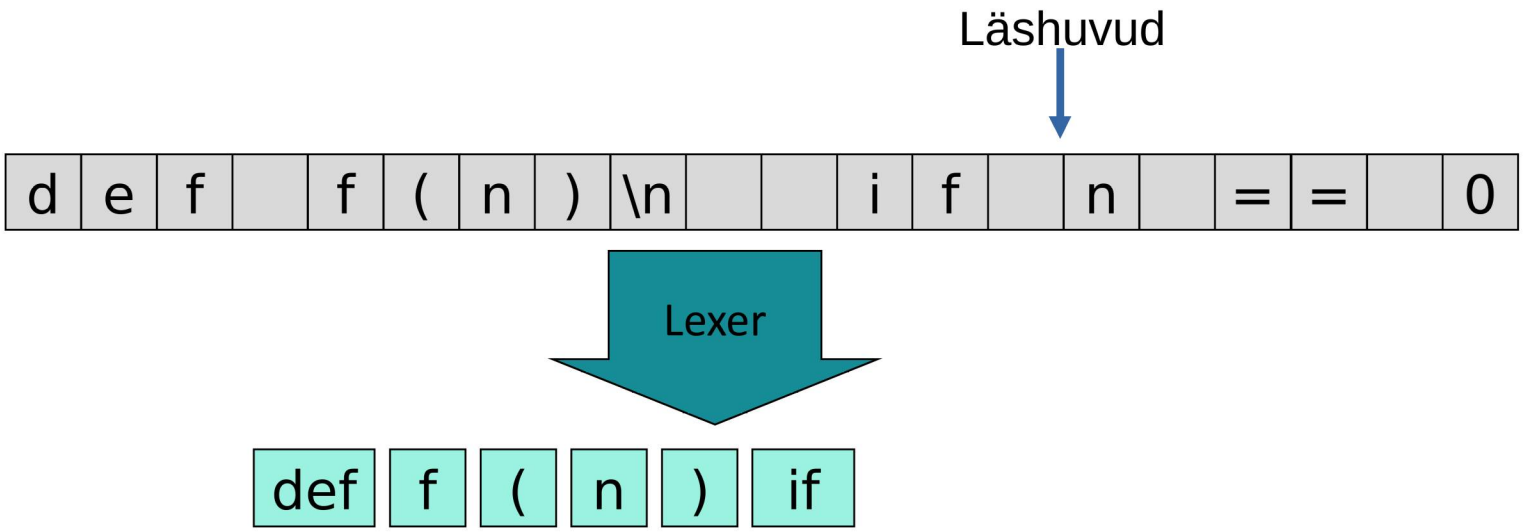
Lexer – Hur görs valen av tokens? (Regexp)



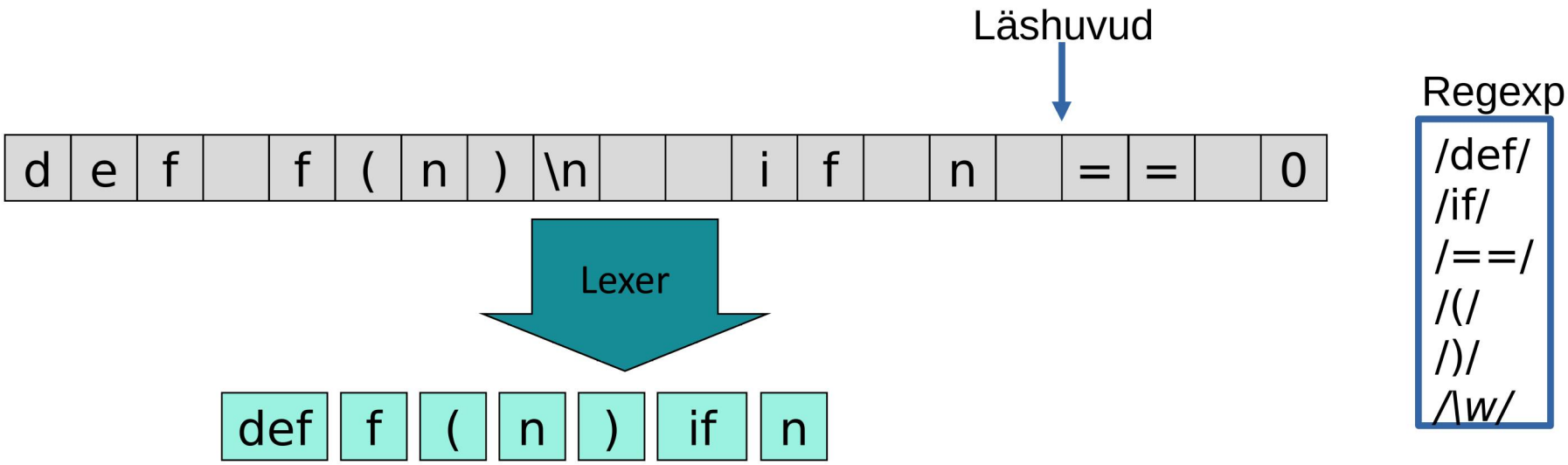
Lexer – Hur görs valen av tokens? (Regexp)



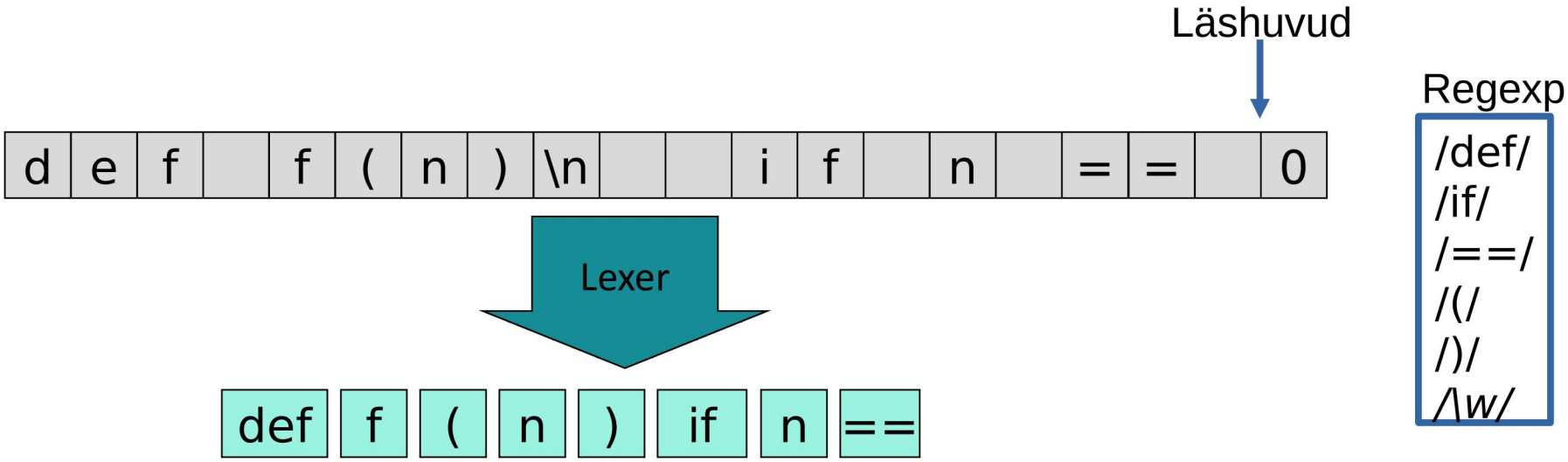
Lexer – Hur görs valen av tokens? (Regexp)



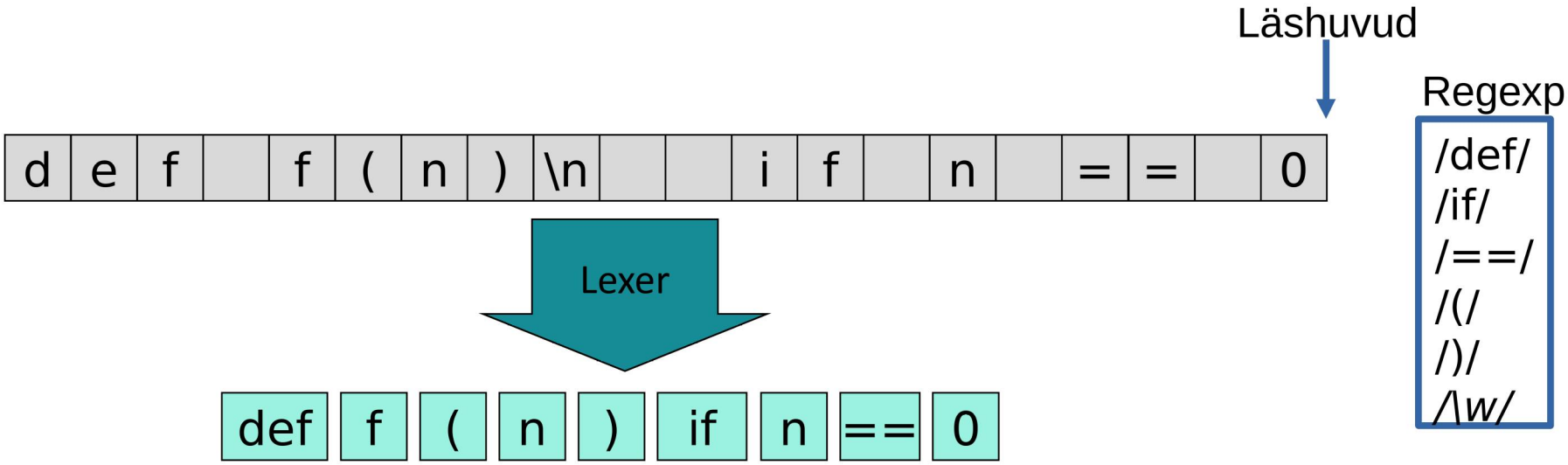
Lexer – Hur görs valen av tokens? (Regexp)



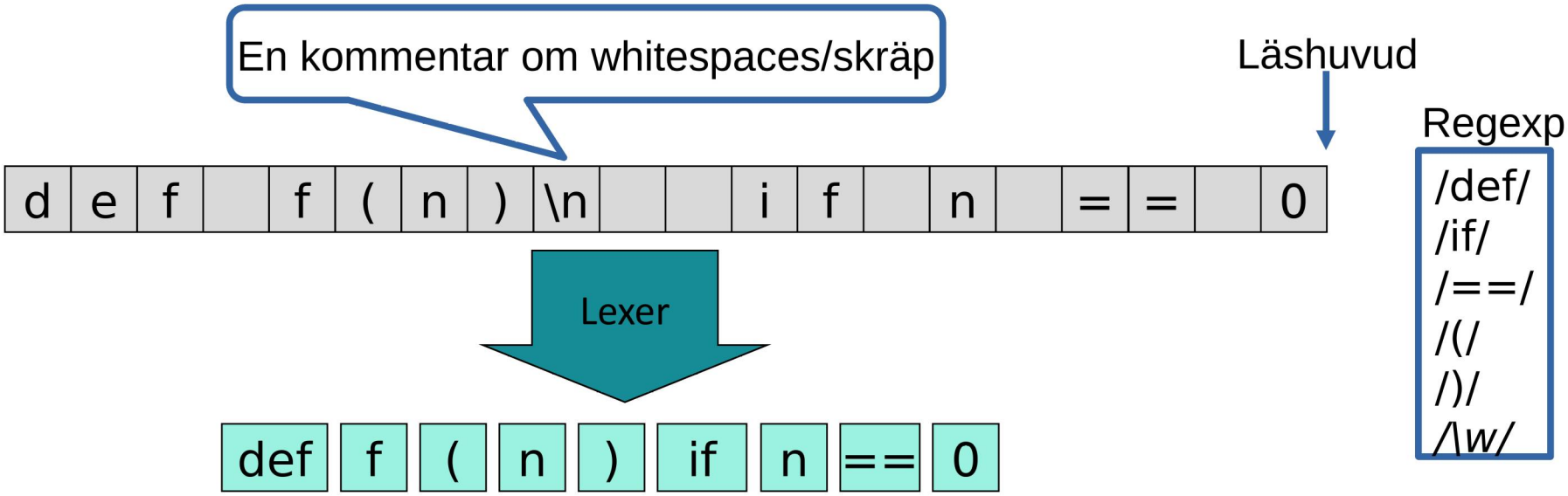
Lexer – Hur görs valen av tokens? (Regexp)



Lexer – Hur görs valen av tokens? (Regexp)



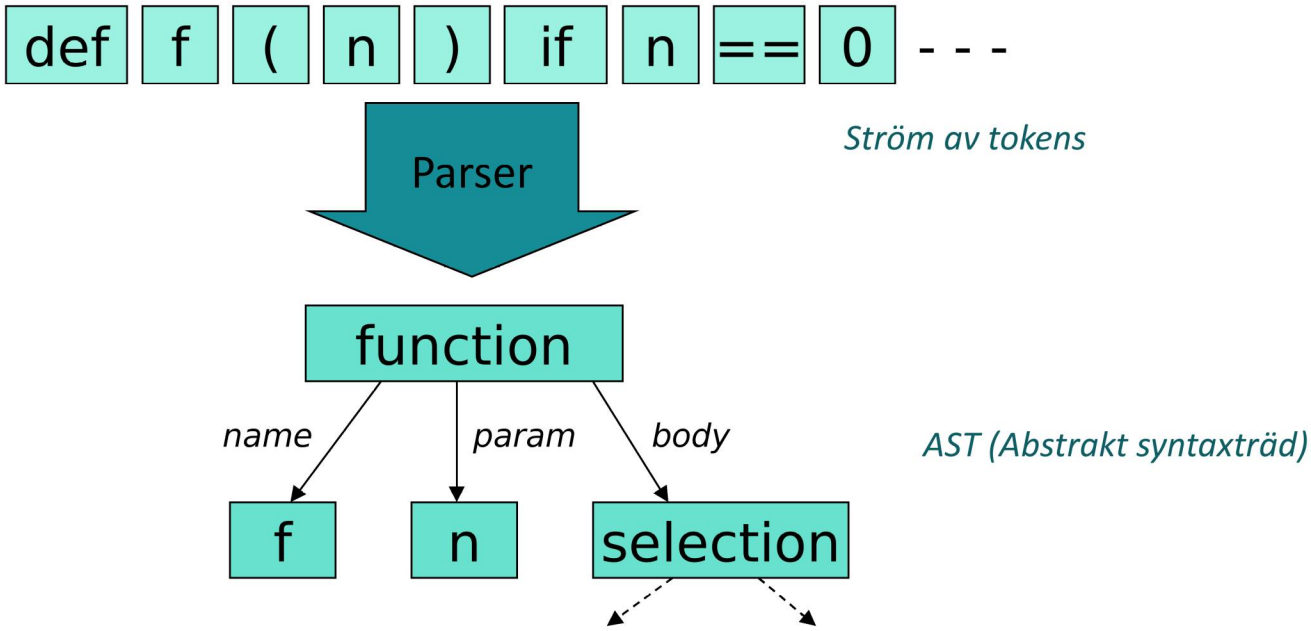
Lexer – Hur görs valen av tokens? (Regexp)

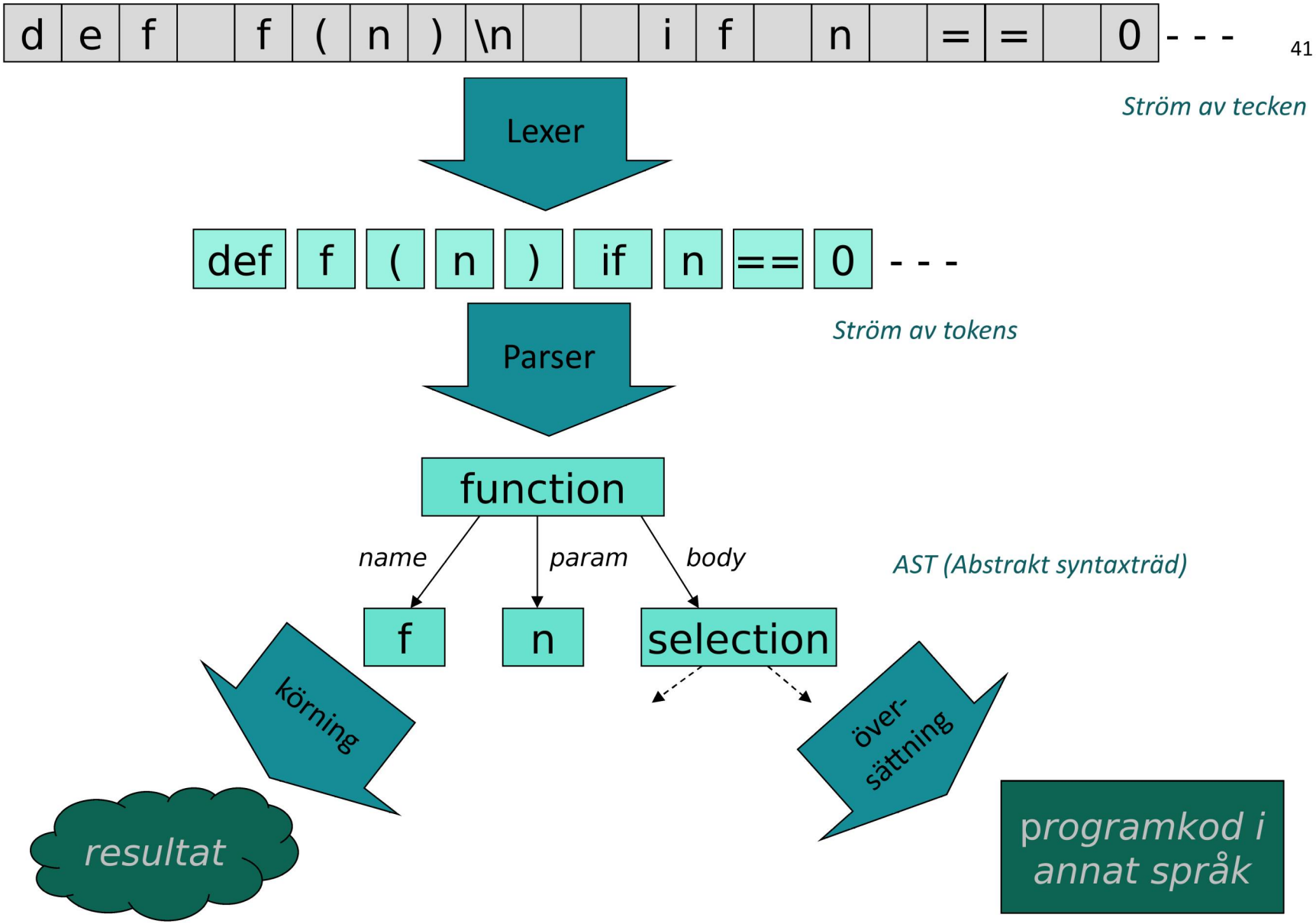


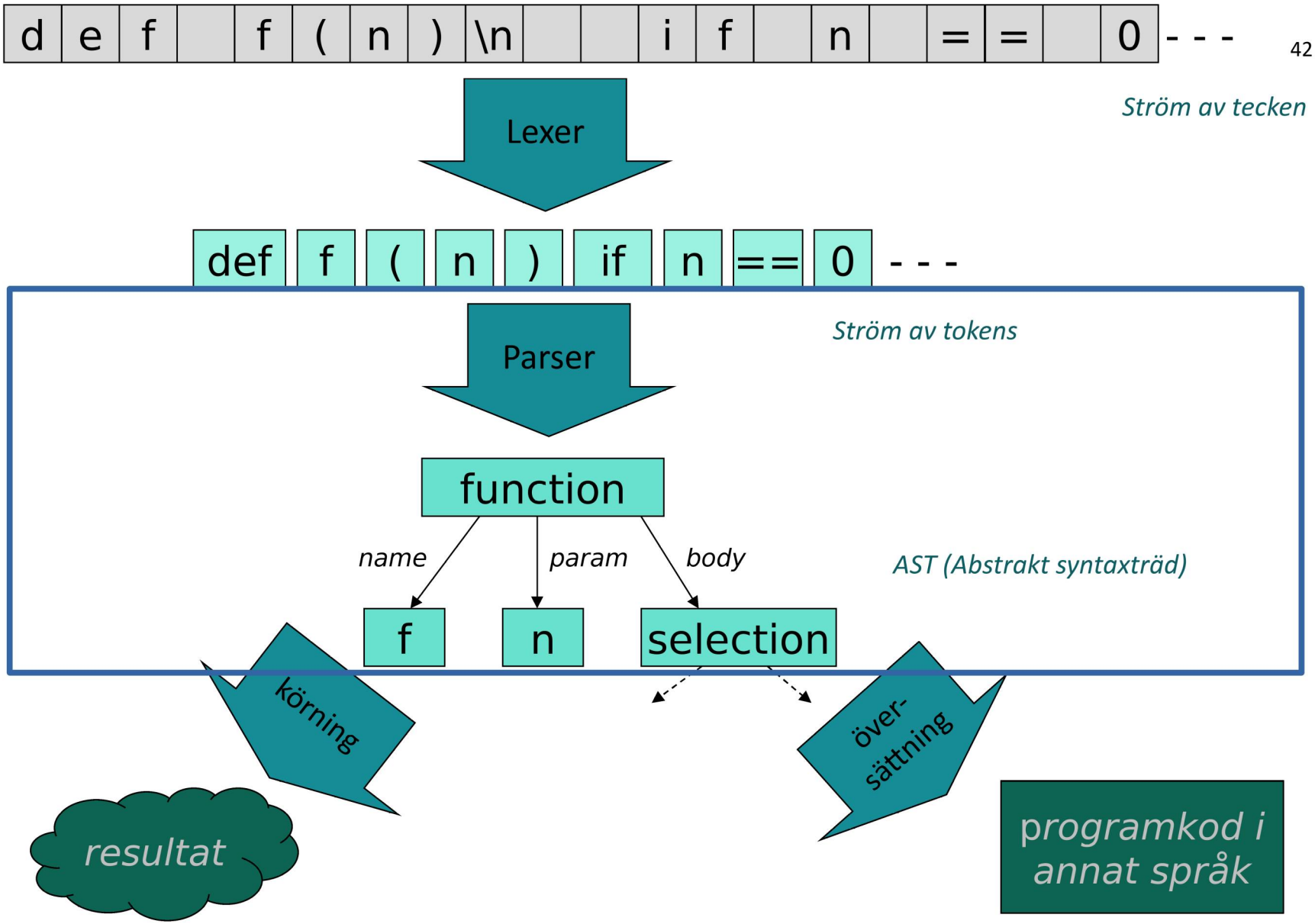
Parser – Hur tillskrivs tokens mening?

Behöver någon sätt att definerar hur tokens ska tolkas

Ofta BNF




d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
f
n
selection
AST (Abstrakt syntaxträd)
körning
över-
sättning
resultat
programkod i
annat språk


d e f f (n) \n i f n = = 0 - - -
Ström av tecken
Lexer
def f (n) if n == 0 - - -
Ström av tokens
Parser
function
name
param
body
AST (Abstrakt syntaxträd)
f
n
selection
körning
över-
sättning
resultat
programkod i
annat språk

Ett exempel på aritmetik och BNF

1 + (2 * 3)

Ett exempel på aritmetik och BNF

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)

1 + (2 * 3)

Ett exempel på aritmetik och BNF

Innifrån → ut

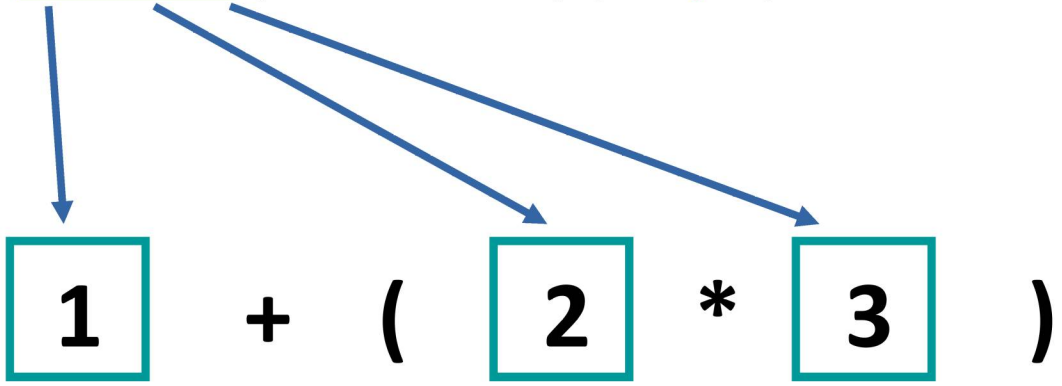
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)

1 + (2 * 3)

Ett exempel på aritmetik och BNF

Innifrån → ut

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)

Ett exempel på aritmetik och BNF

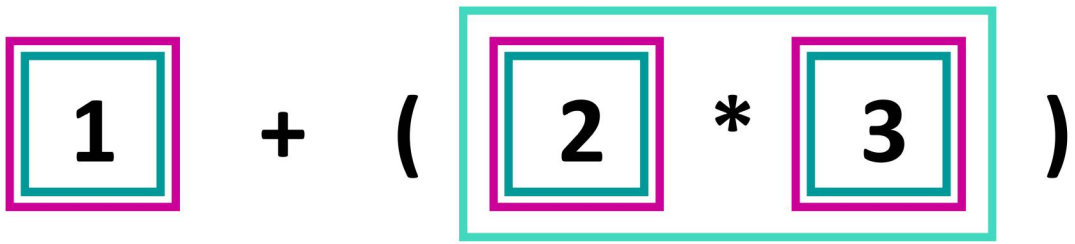
Innifrån → ut

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)

Ett exempel på aritmetik och BNF

Innifrån → ut

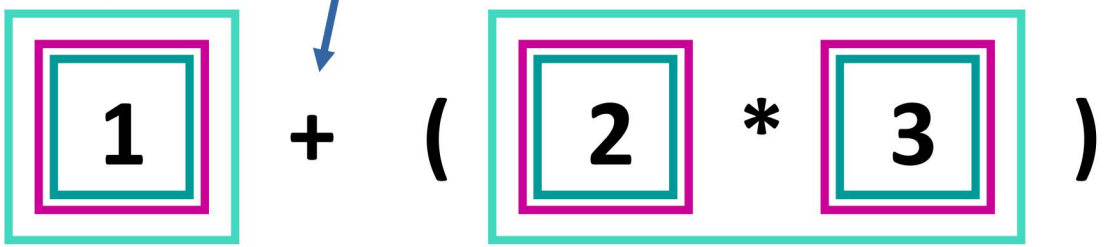
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

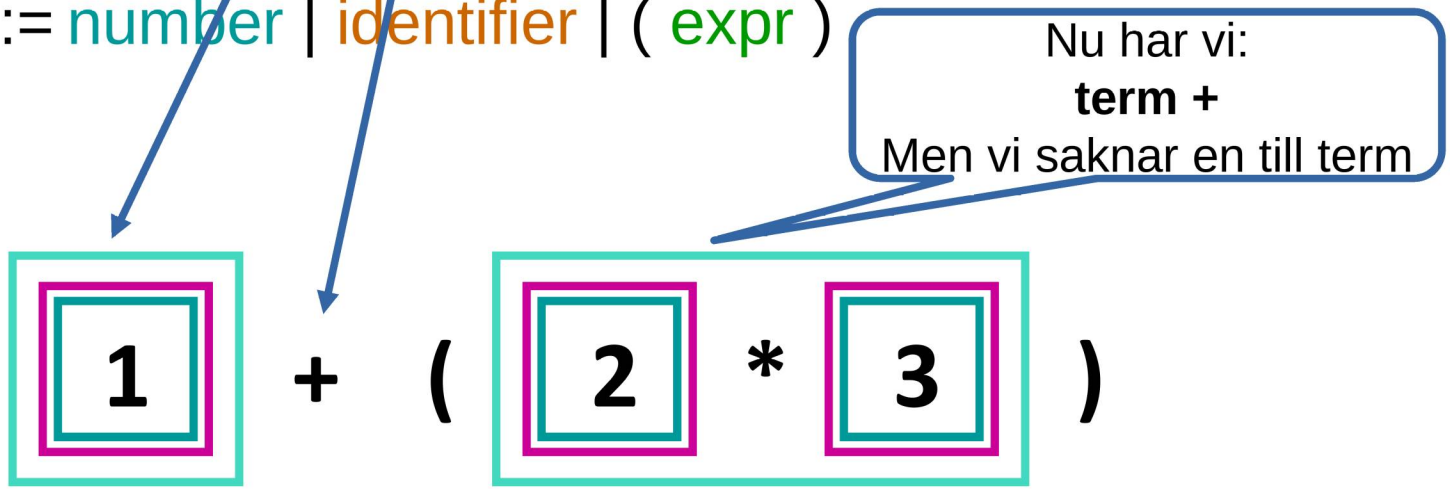
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

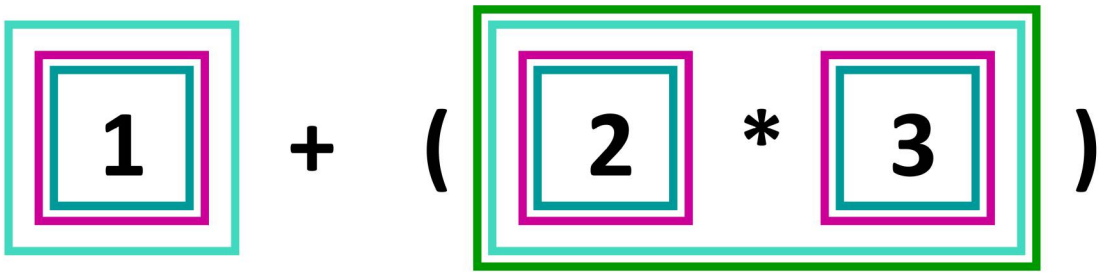
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

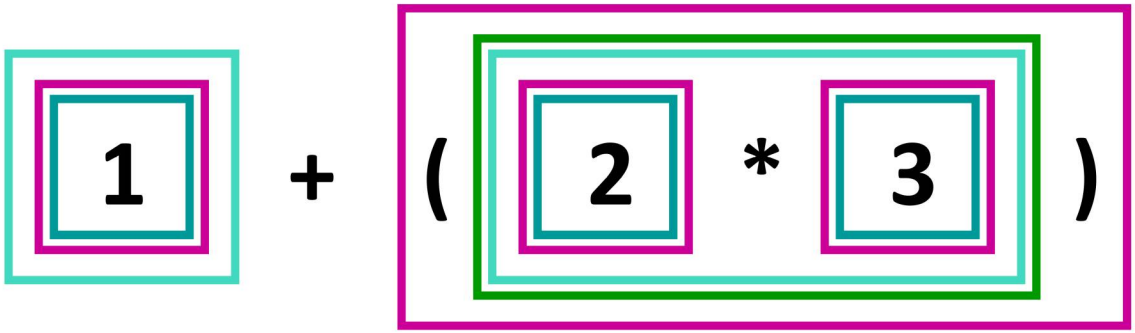
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

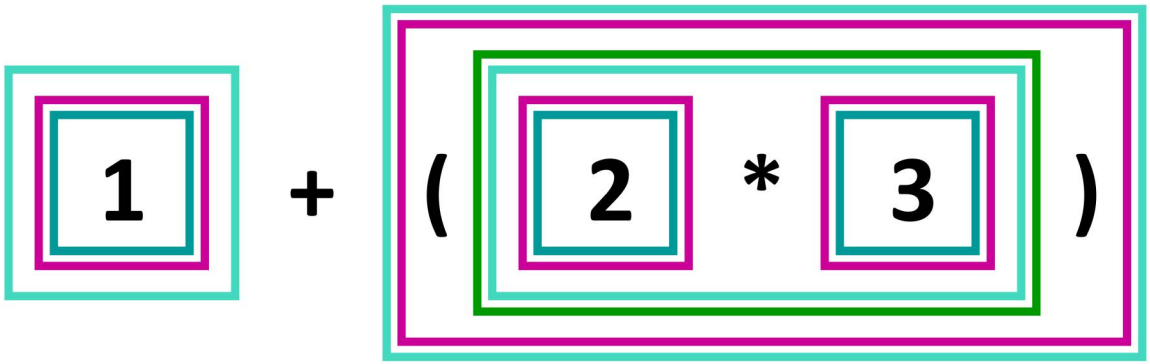
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

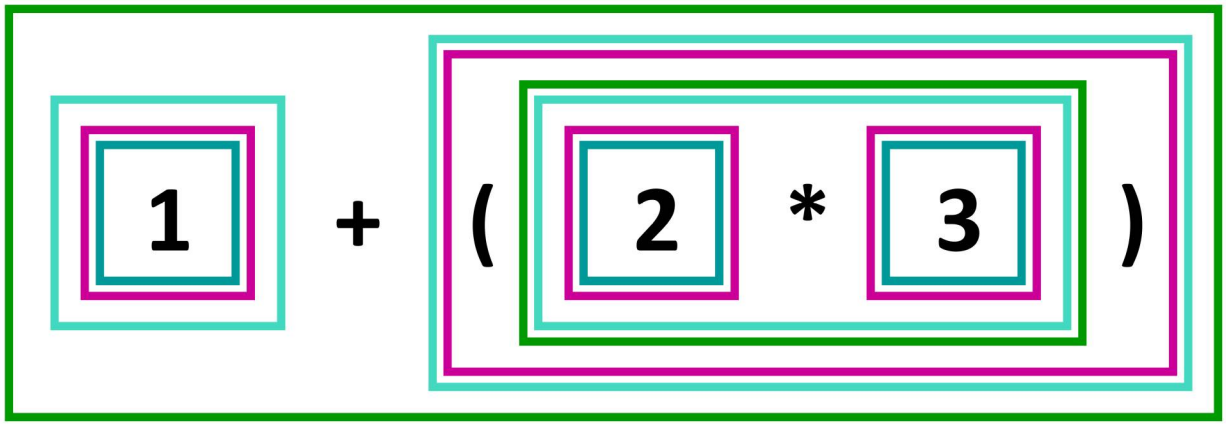
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Innifrån → ut

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

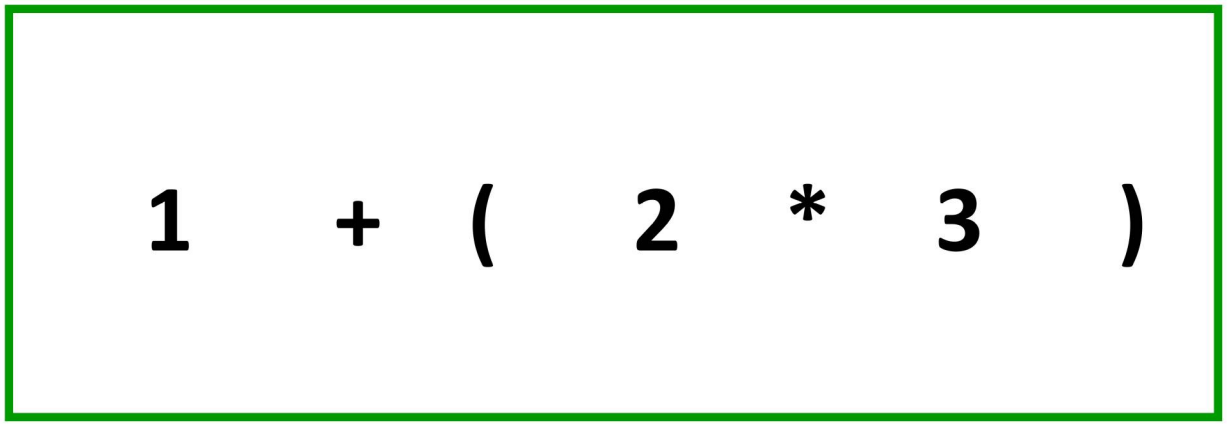
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)

1 + (2 * 3)

Ett exempel på aritmetik och BNF

Utifrån → in

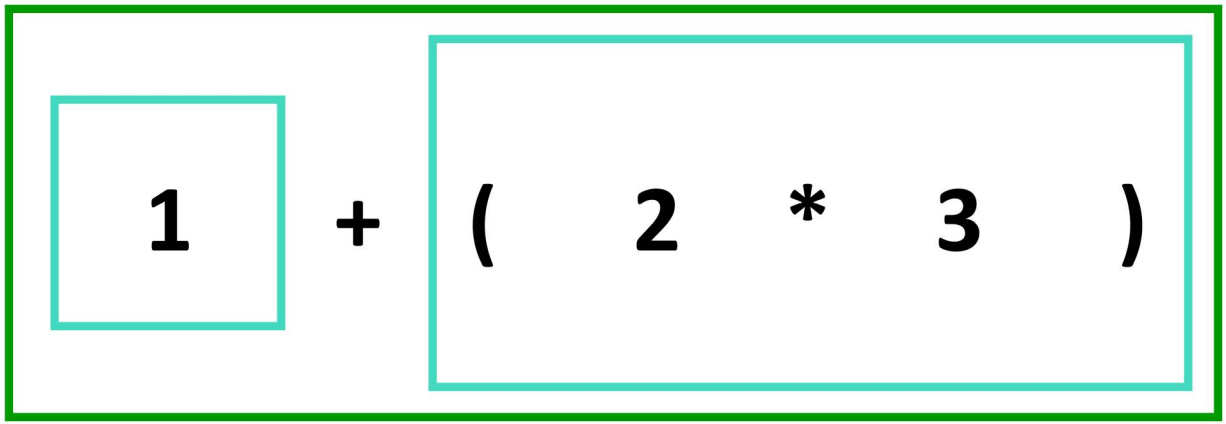
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

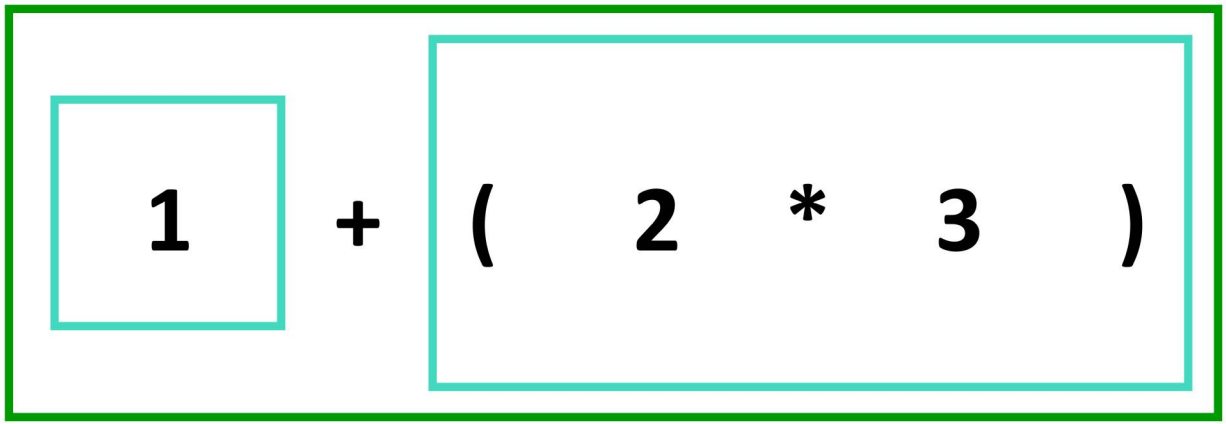
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

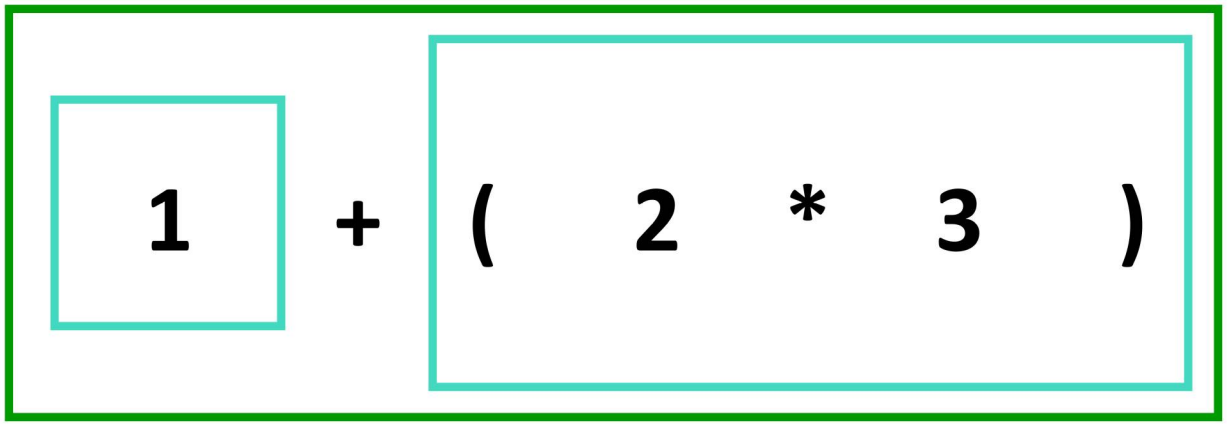
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

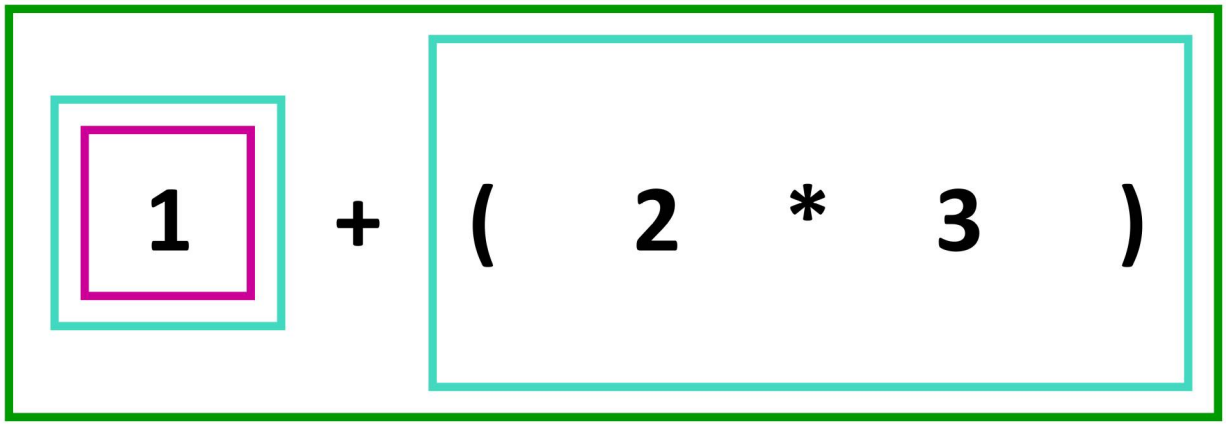
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

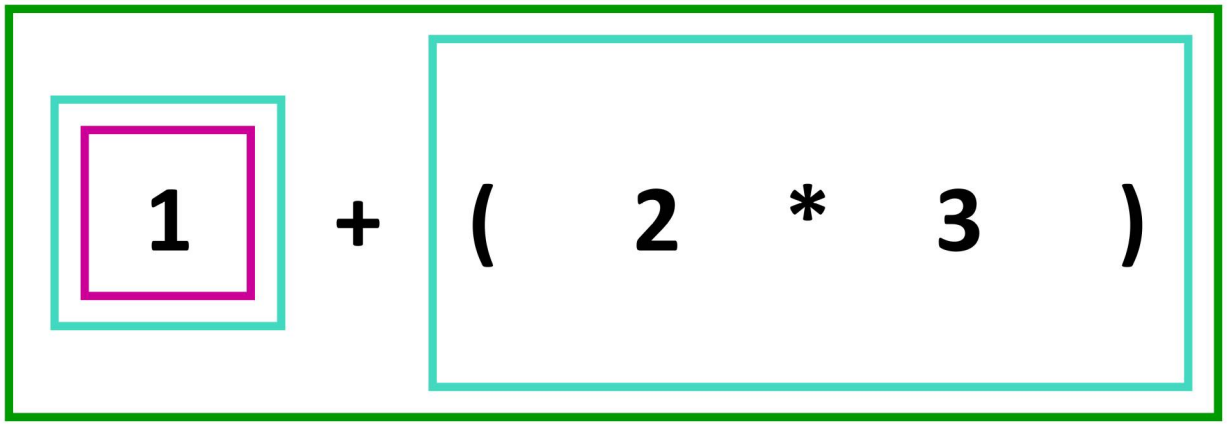
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

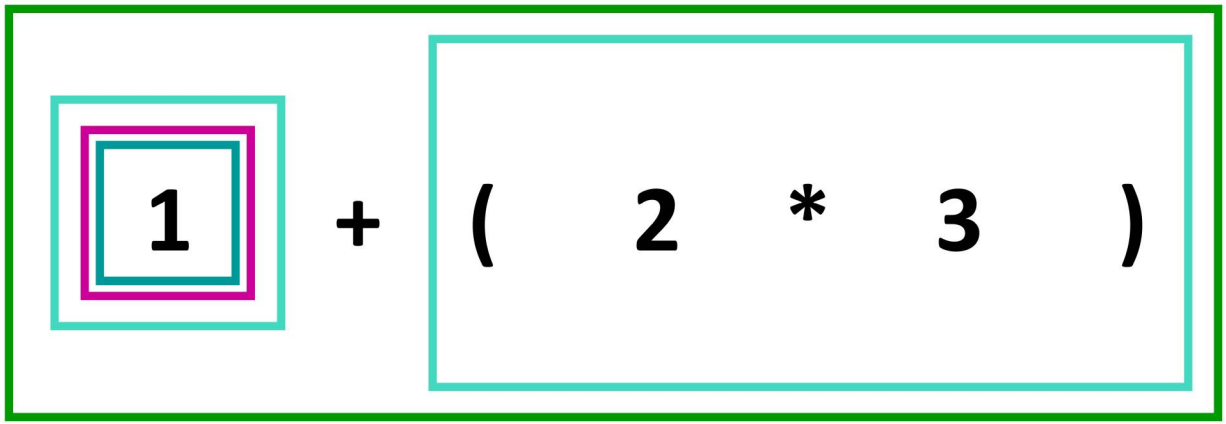
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

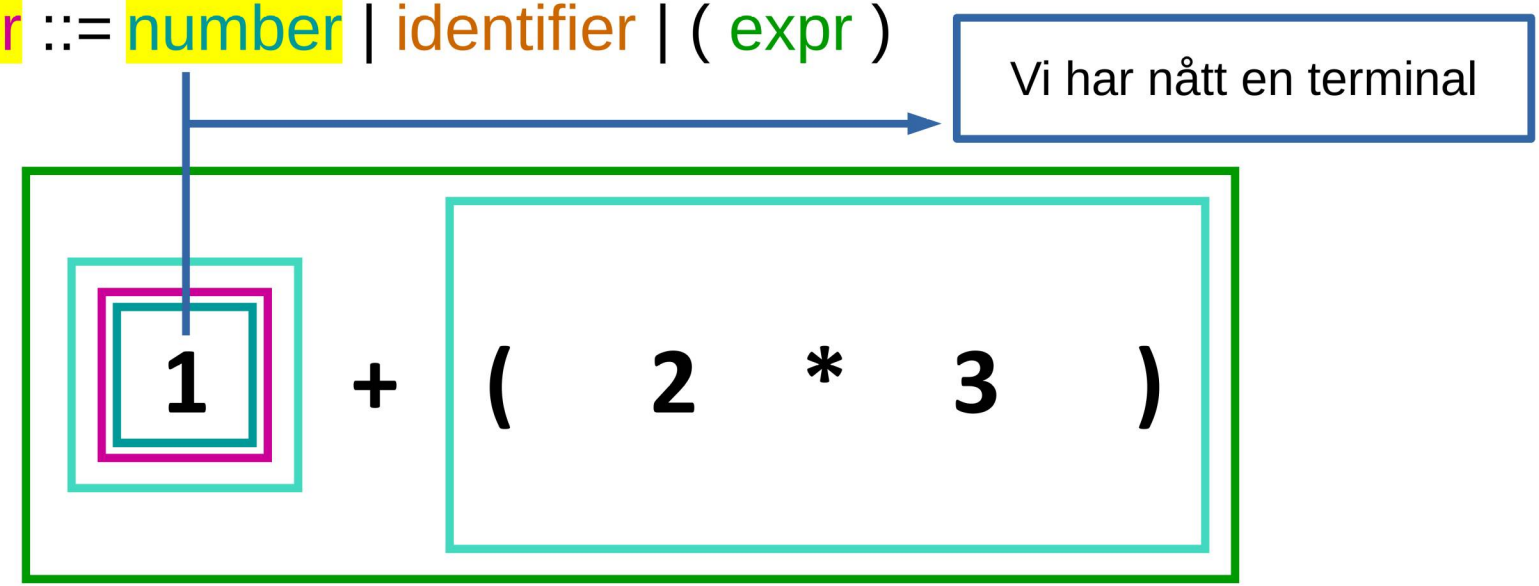
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

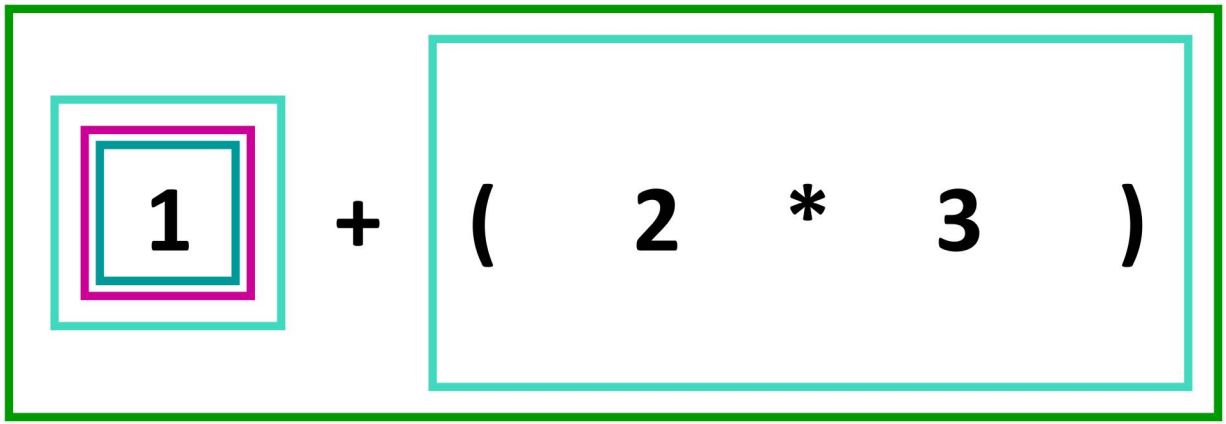
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

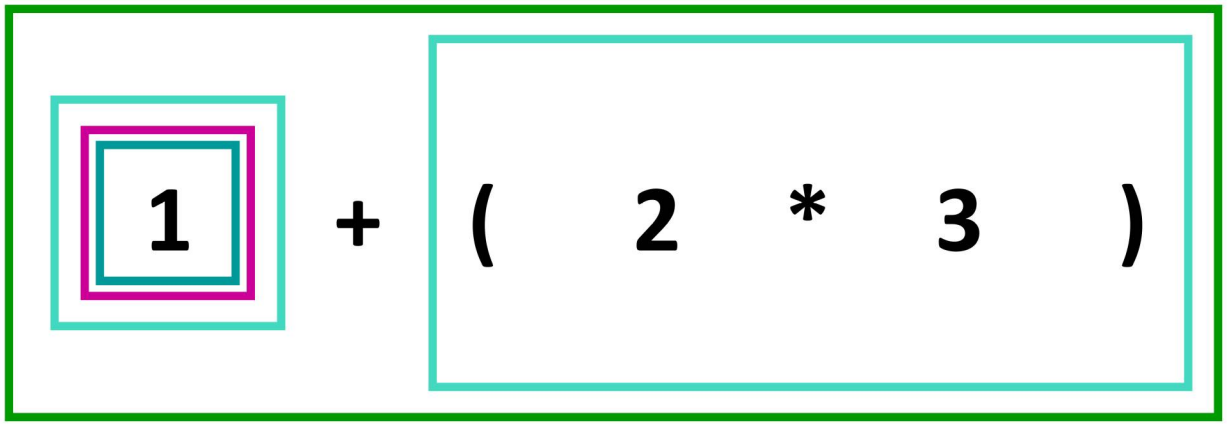
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

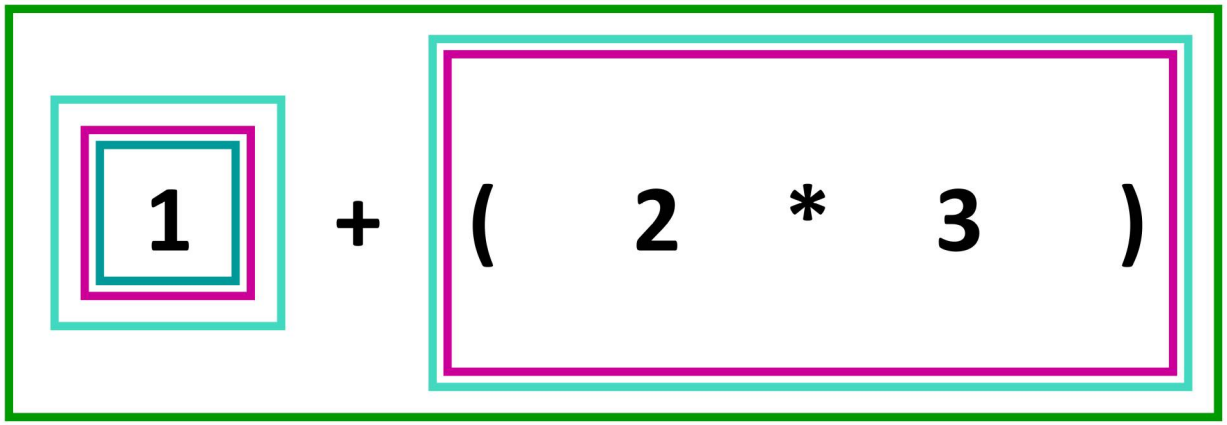
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

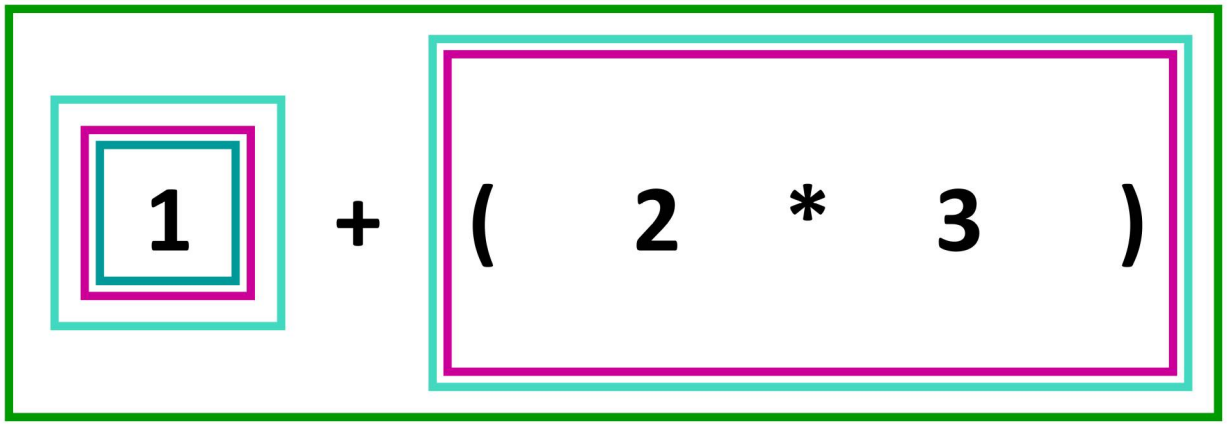
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

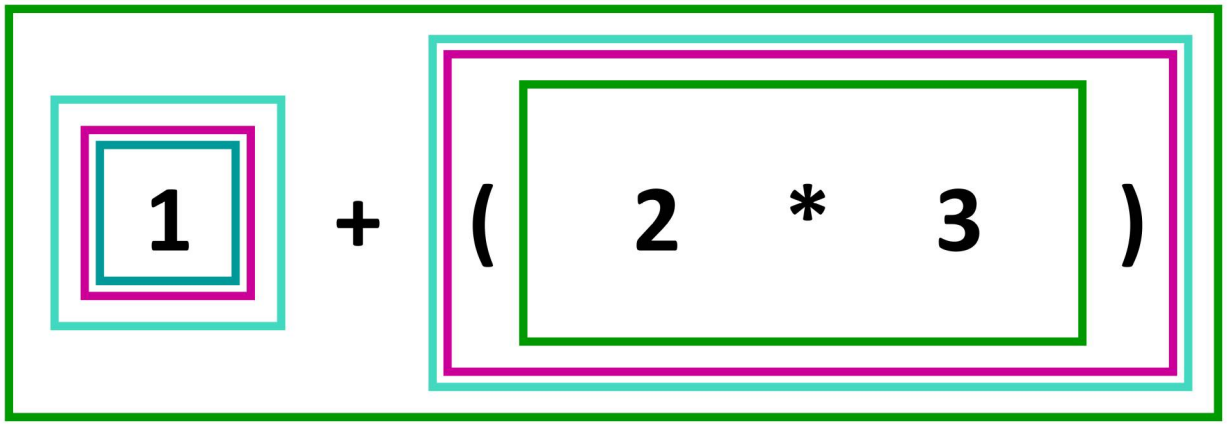
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

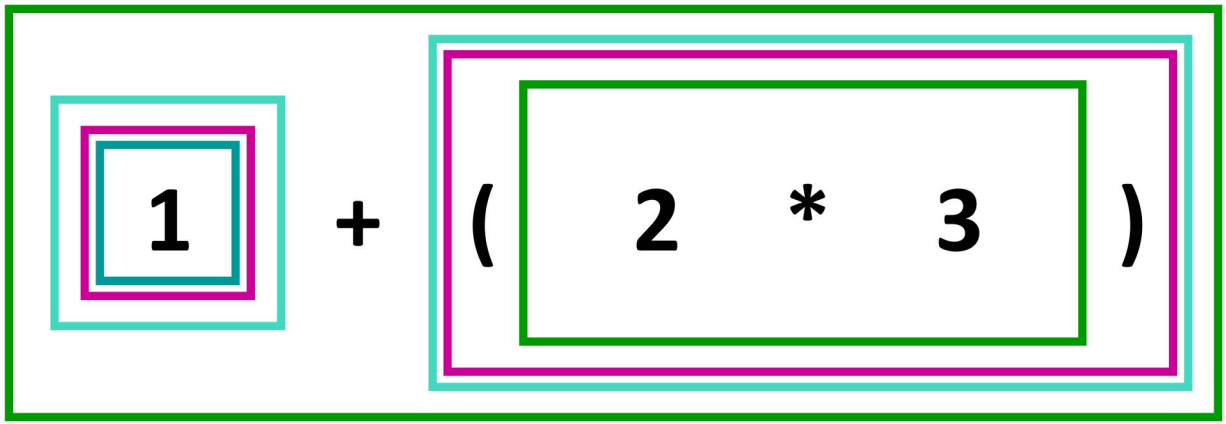
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

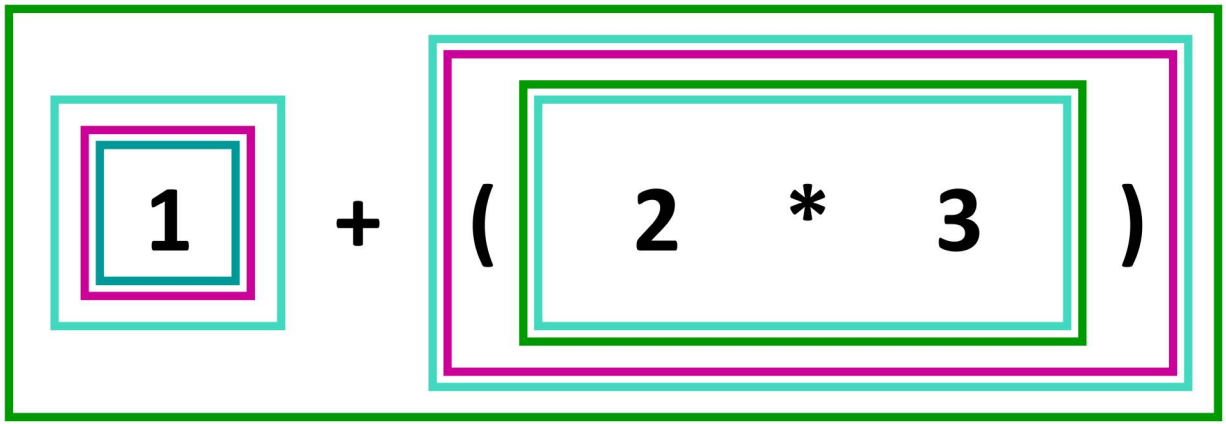
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

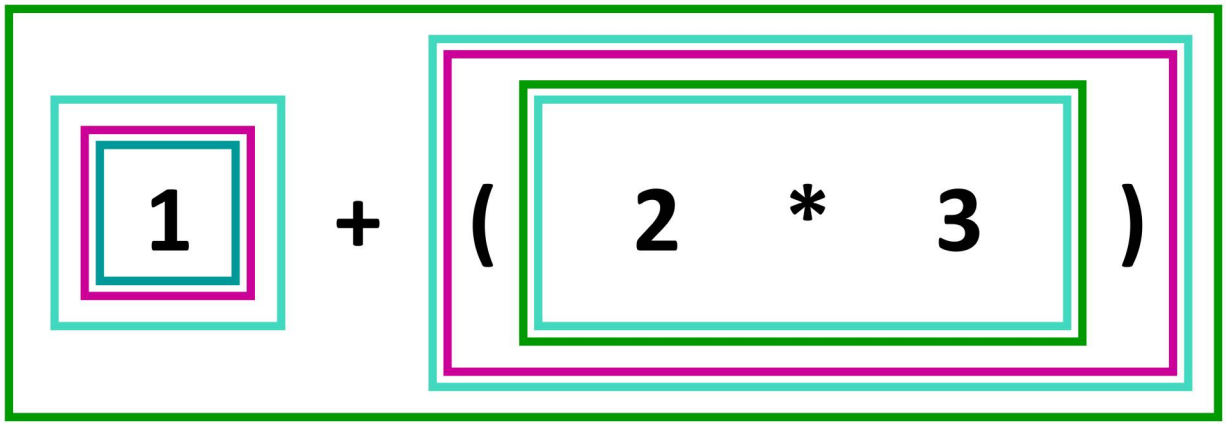
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



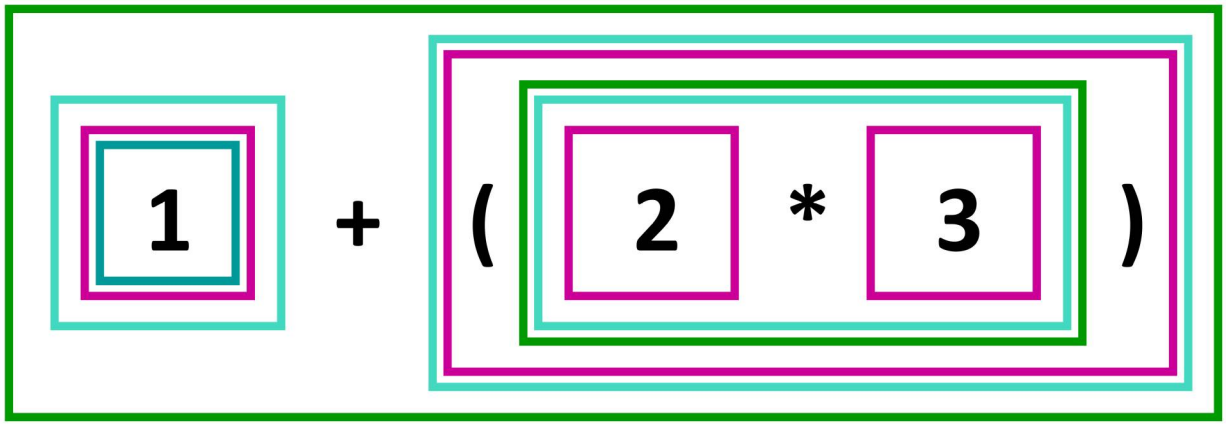
Ett exempel på aritmetik och BNF

Utifrån → in

expr ::= term | term + term | term – term

term ::= factor | factor * factor | factor / factor

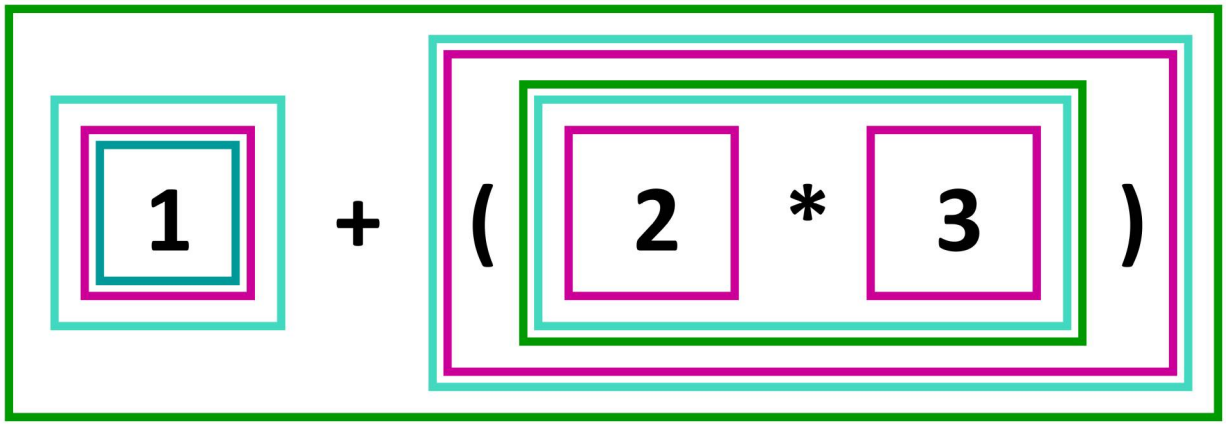
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

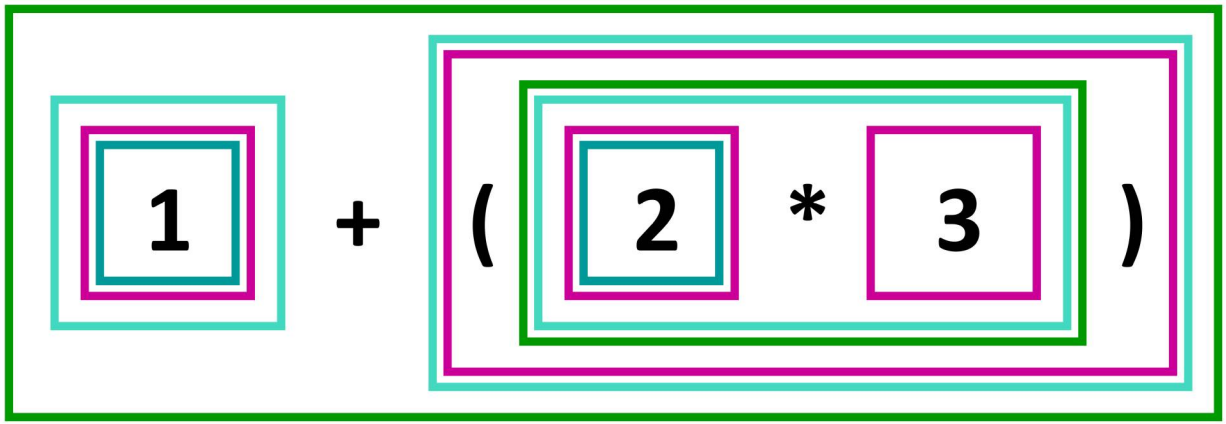
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

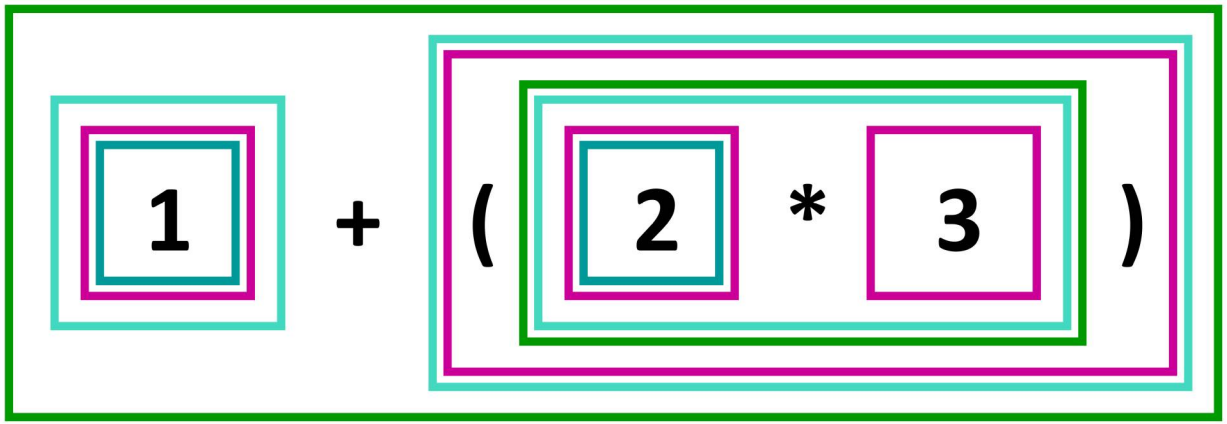
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

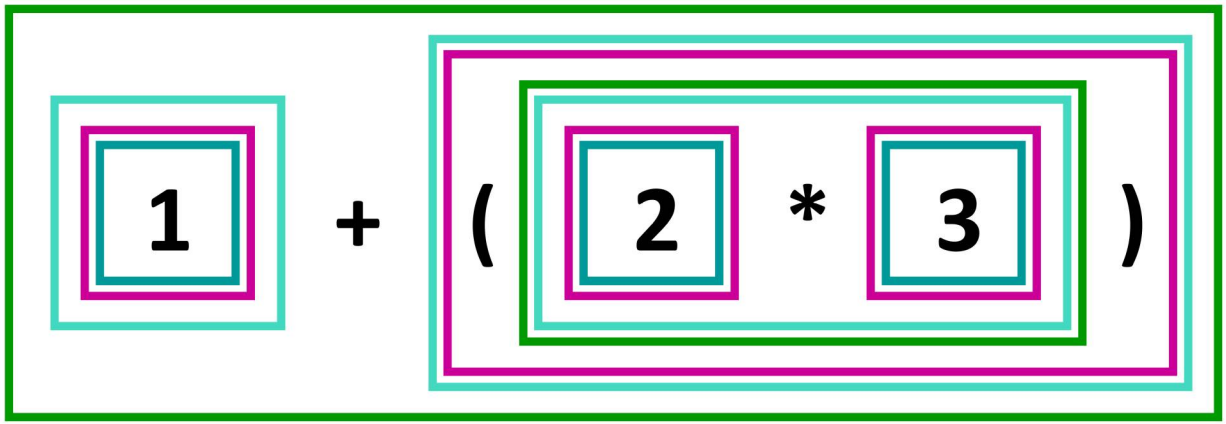
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

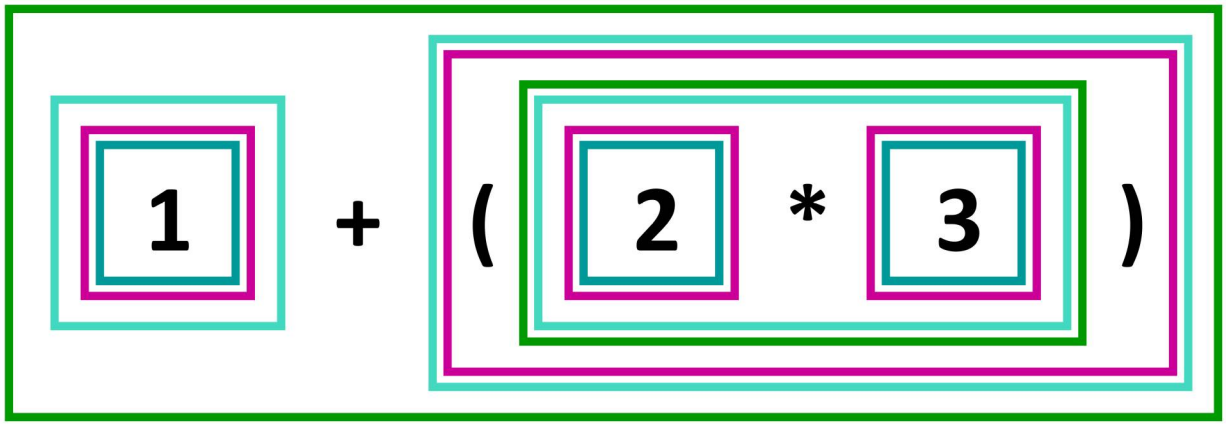
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Ett exempel på aritmetik och BNF

Utifrån → in

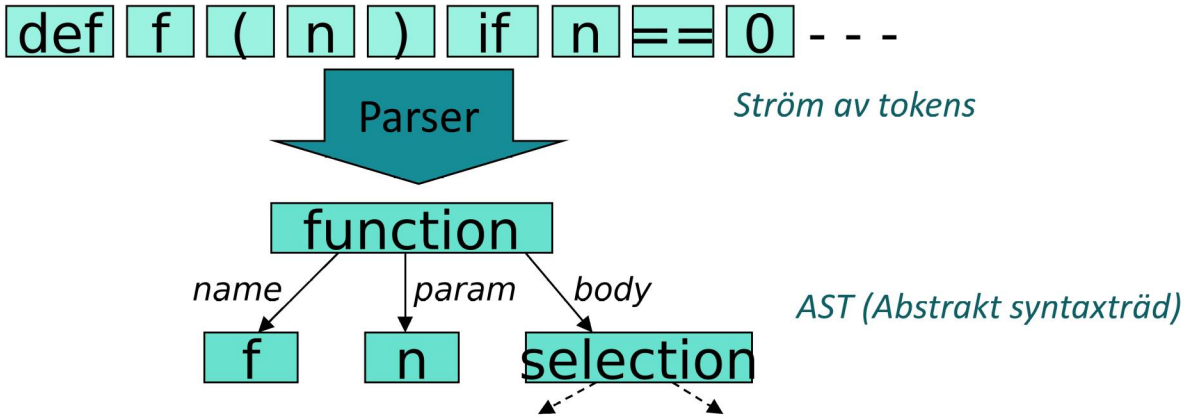
expr ::= term | term + term | term – term
term ::= factor | factor * factor | factor / factor
factor ::= number | identifier | (expr)



Parser – Hur tillskrivs tokens mening?

Så parsern kan använda en grammatik för att kontrollera att den givna kombinationen av tokens är syntaktiskt korrekt.

Vi kommer kunna ingripa vid matchningen av dessa olika regler för att tillskriva mening till vår samling av tokens. Det här är hur vi bestämmer vad som händer när vi skriver ”1 + 2”



Verktyg för generering av verktyg

- Klassiska Unix-verktyg för C:
  - *lex* är ett verktyg för att skapa lexers, där man specificerar språkets lexikaliska struktur med reguljära uttryck.
  - *yacc* är ett verktyg för att skapa parsers, där man specificerar språkets syntax med en kontextfri grammatik (ofta uttryck med BNF).
- Det finns motsvarande verktyg för Ruby också, t.ex. ruby-lex och racc.

Parsern som vi använder

- Två basklasser:
  - *Rule* representerar en syntaktisk regel.
  - *Parser* representerar själva parsern (generisk).
- Med hjälp av dessa skapar vi ett litet *domänspecifikt språk* så att vi lätt kan göra en egen parser för ett valfritt språk.
- I labbfilerna finns exempelparsern *DiceRoller* som är ett litet språk för att slå tärningar och räkna ut enkla matematiska uttryck.
- Alla tre klasserna *Rule, Parser* och *DiceRoller* finns i filen rdparse.rb.
- Detta är en *Recursice Descent Parser,* därav namnet.

Användning av DiceRoller-exemplet

```
irb(main):1696:0>
DiceRoller.new.roll
[diceroller] 1+3
=> 4
[diceroller] 1+1d4
=> 2
[diceroller] 1+1d4
=> 3
[diceroller] (2+8*1d20)*3d6
=> 306
```

Slå en tärning med fyra sidor och lägg till ett.

Slå en tärning med tjugo sidor, multiplicera med åtta och lägg till två. Multiplicera detta med summan av tre tärningar med sex sidor.

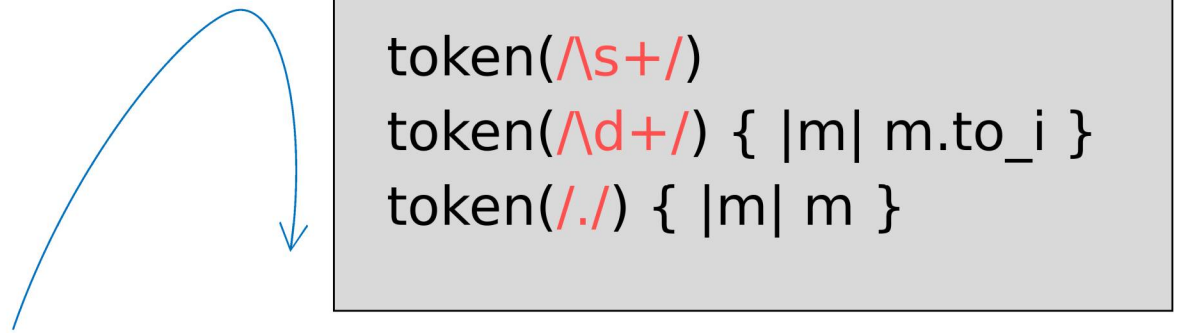
Övning

- Hämta filen rdparse.rb, titta på koden och testkör *DiceRoller*-språket.
- **Fokusera på klassen DiceRoller**
- Var styrs beteende för multiplikation etc?
- Ändra något beteende (hur addition eller dyl fungerar).

Lexer – från sträng till tokens

```
token(/\s+/)
token(/\d+/) { |m| m.to_i }
token(/./) { |m| m }
```

Lexer – från sträng till tokens



Matchningen görs uppifrån och ner (och från vänster till höger i strängen)

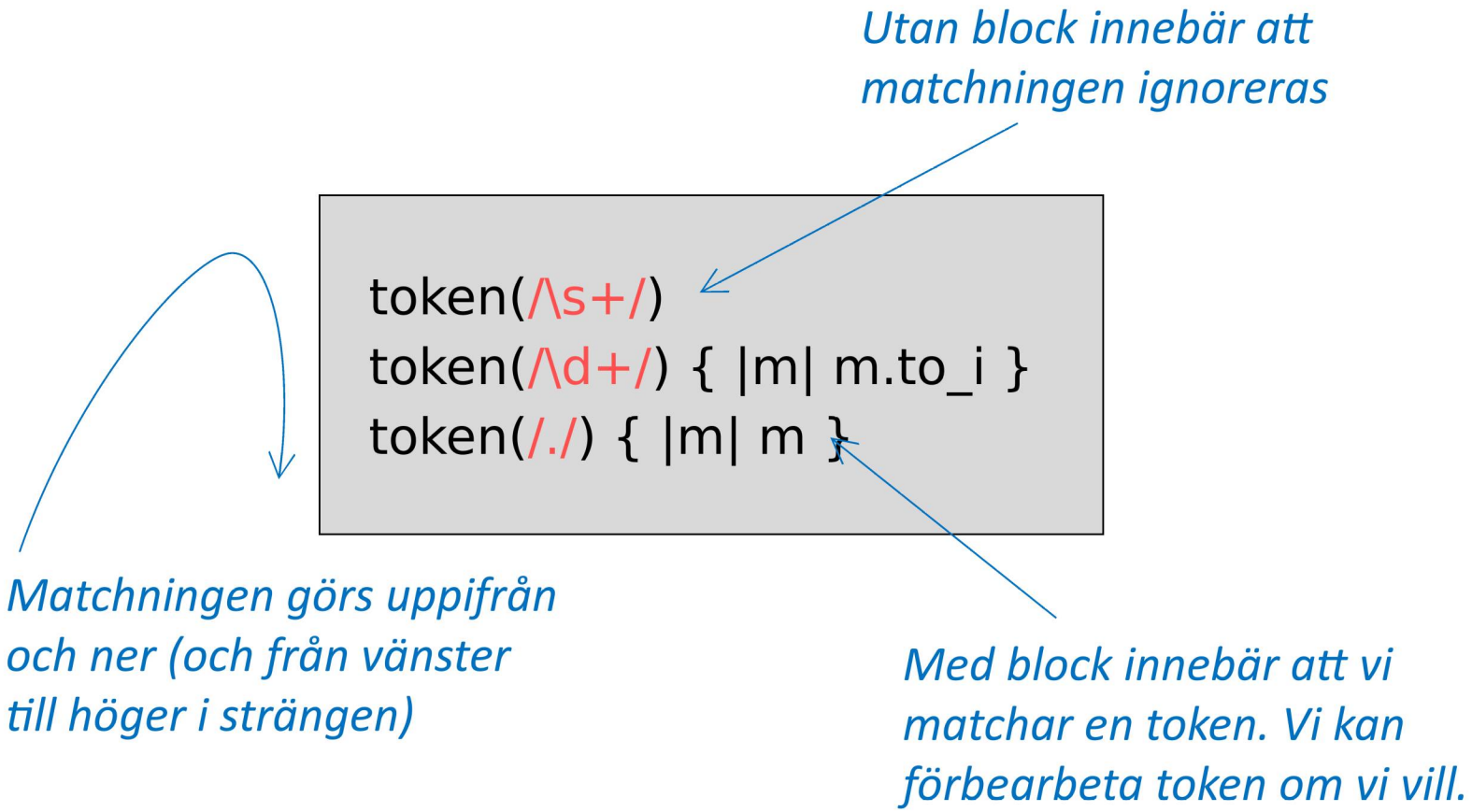
Lexer – från sträng till tokens

Utan block innebär att matchningen ignoreras

```
token(/\s+/)
token(/\d+/) { |m| m.to_i }
token(/./) { |m| m }
```

Matchningen görs uppifrån och ner (och från vänster till höger i strängen)

Lexer – från sträng till tokens

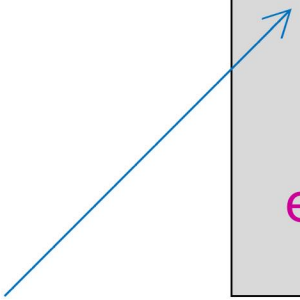


Parser – tolkning av tokens

```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```

Parser – tolkning av tokens

```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```



*En **:term** är något som*
matchar en av dessa tre regler.

Parser – tolkning av tokens

*Om vi ser en **:term**, en * och en **:dice** i följd...*

```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```

*En **:term** är något som matchar en av dessa tre regler.*

Parser – tolkning av tokens

*Om vi ser en **:term**, en * och en **:dice** i följd...*

*... då beräknar vi vad **:term** gånger **:dice** är.*

```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```

*En **:term** är något som matchar en av dessa tre regler.*

Parser – tolkning av tokens

*Om vi ser en **:term**, en * och en **:dice** i följd...*

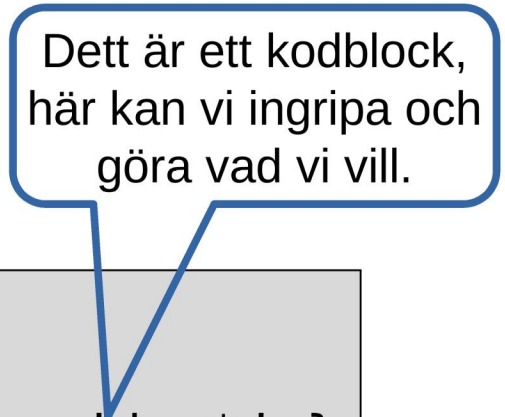
*... då beräknar vi vad **:term** gånger **:dice** är.*

```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```

*En **:term** är något som matchar en av dessa tre regler.*

*Om vi ser en **:dice** returnerar vi den som den är.*

Parser – tolkning av tokens




```
rule :term do
  match(:term, '*', :dice) { |a, _, b| a * b }
  match(:term, '/', :dice) { |a, _, b| a / b }
  match(:dice)
end
```

```
class DiceRoller

  def initialize
    @diceParser = Parser.new("dice roller") do
      token(/\s+/)
      token(/\d+/) {|m| m.to_i }
      token(/./) {|m| m }

      start :expr do
        match(:expr, '+', :term) {|a, _, b| a + b }
        match(:expr, '-', :term) {|a, _, b| a - b }
        match(:term)
      end

      rule :term do
        match(:term, '*', :dice) {|a, _, b| a * b }
        match(:term, '/', :dice) {|a, _, b| a / b }
        match(:dice)
      end
```

lexer

parser

```
    rule :dice do
      match(:atom, 'd', :sides) {|a, _, b| DiceRoller.roll(a, b) }
      match('d', :sides) {|_, b| DiceRoller.roll(1, b) }
      match(:atom)
    end

    rule :sides do
      match('%') { 100 }
      match(:atom)
    end

    rule :atom do
      match(Integer)
      match('(', :expr, ')') {|_, a, _| a }
    end
  end
 end

end
```

Övning

- Ändra *DiceRoller* så att den, istället för att faktiskt beräkna uttrycken direkt, returnerar någon form av *abstrakt syntaxträd*. Ni får själva bestämma hur det ska se ut (arrayer, hashtabeller, egna klasser, …).

BNF-grammatik för DiceRoller-språket

expr ::= *expr* **+** *term* | *expr* **-** *term* | *term*

term ::= *term* * *dice* | *term* **/** *dice* | *dice*

dice ::= *atom* **d** *sides* | **d** *sides* | *atom*

sides ::= **%** | *atom*

atom ::= Integer | (*expr*)

Inför seminarie 3

- Tekniker, tips, modifikationer - rdparse
- En tydlig förankring i tidigare programspråk, funkar allt det här vi gör att göra även i c++/python, vad är skillnaderna?
- Vad är överförbart från tidigare språk/projekt till det arbete ni gör nu?

www.liu.se

LINKÖPINGS UNIVERSITET