Domänspecifika språk

TDP007 Konstruktion av datorspråk
Föreläsning 5

LINKÖPINGS UNIVERSITET

Pontus Haglund
Institutionen för datavetenskap
2024-01-23

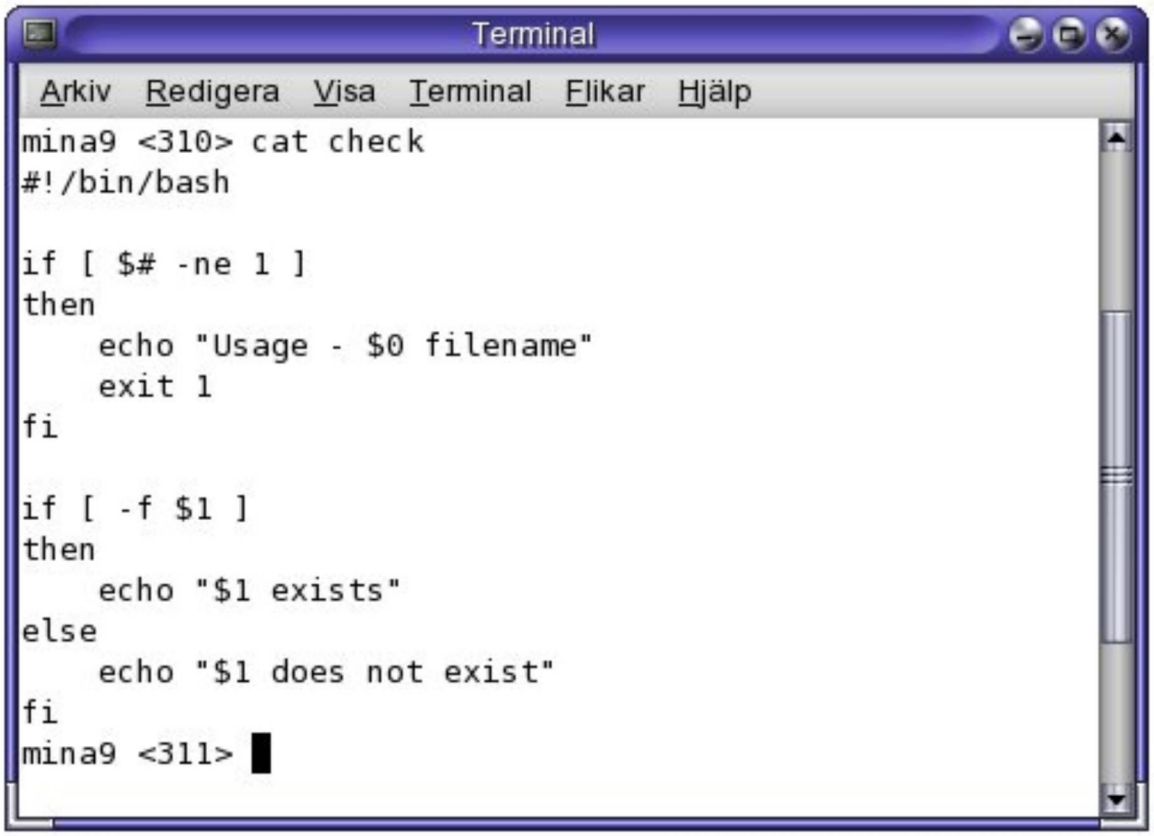
Du kan nu:

- Logga in via thinlinc
- Hämta följande filer från föreläsningssidan:
  - warehouse.rb
  - lager.txt
  - expr.rb
- Vi kommer jobba med dessa under FÖ

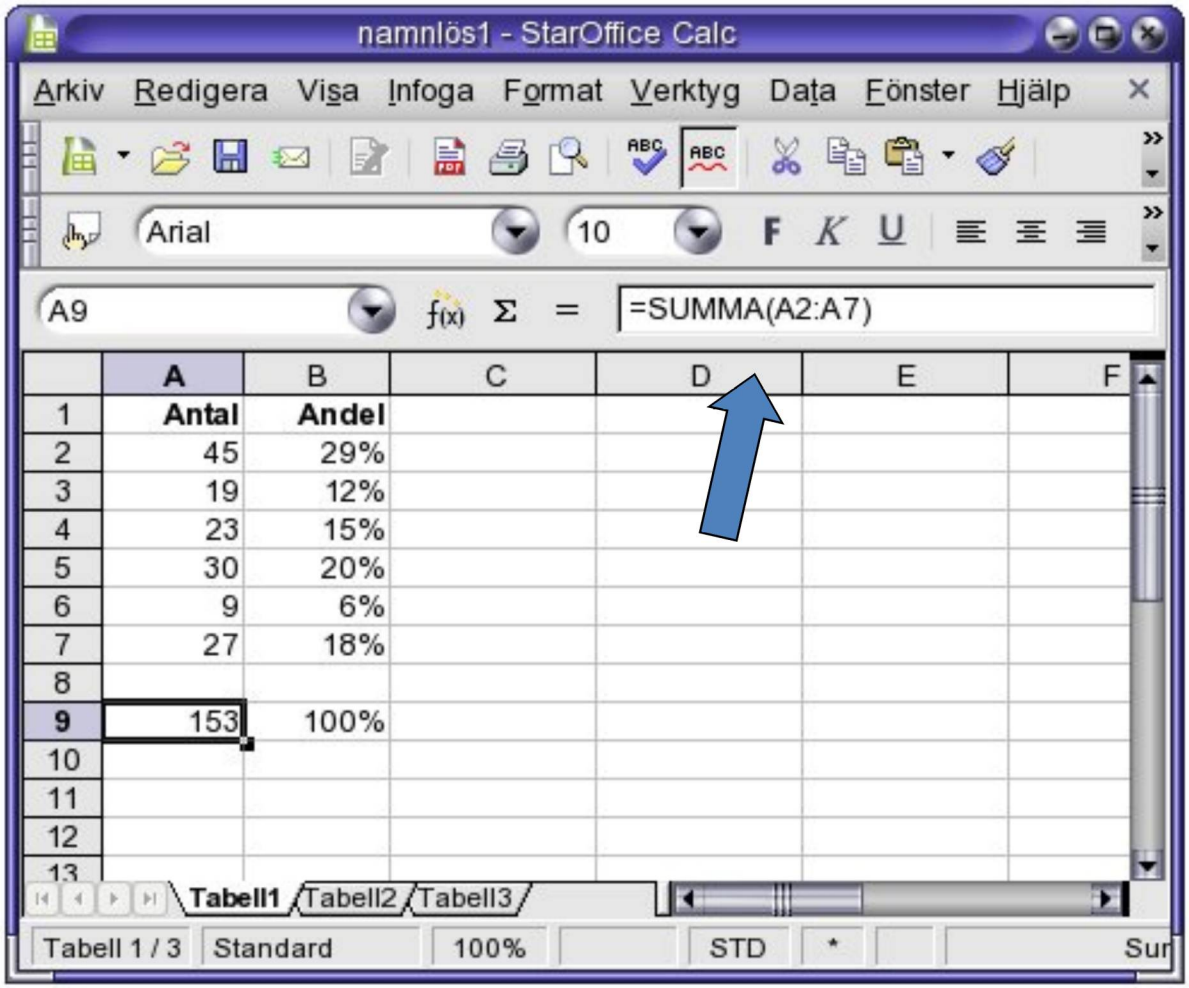
Domänspecifika språk

- Ett *domänspecifikt språk* (eng. domain specific language, DSL) är ett oftast litet språk vars syfte är att uttrycka problem eller lösningssätt för ett specifikt begränsat problemområde.
- Alternativa benämningar: *application oriented language, special purpose language, specialized language, task-specific language*.
- Alla kanske inte är med om att skapa ett fullskaligt programspråk, men det är inte otroligt att man kommer att ta fram ett domänspecifikt språk någon gång under karriären.
- Wikipedia-artikeln om *domain-specific language* är en bra introduktion.

Exempel


Terminal
Arkiv Redigera Visa Terminal Flikar Hjälp
mina9 <310> cat check
#!/bin/bash
if [$# -ne 1]
then
echo "Usage - $0 filename"
exit 1
fi
if [-f $1]
then
echo "$1 exists"
else
echo "$1 does not exist"
fi
mina9 <311>

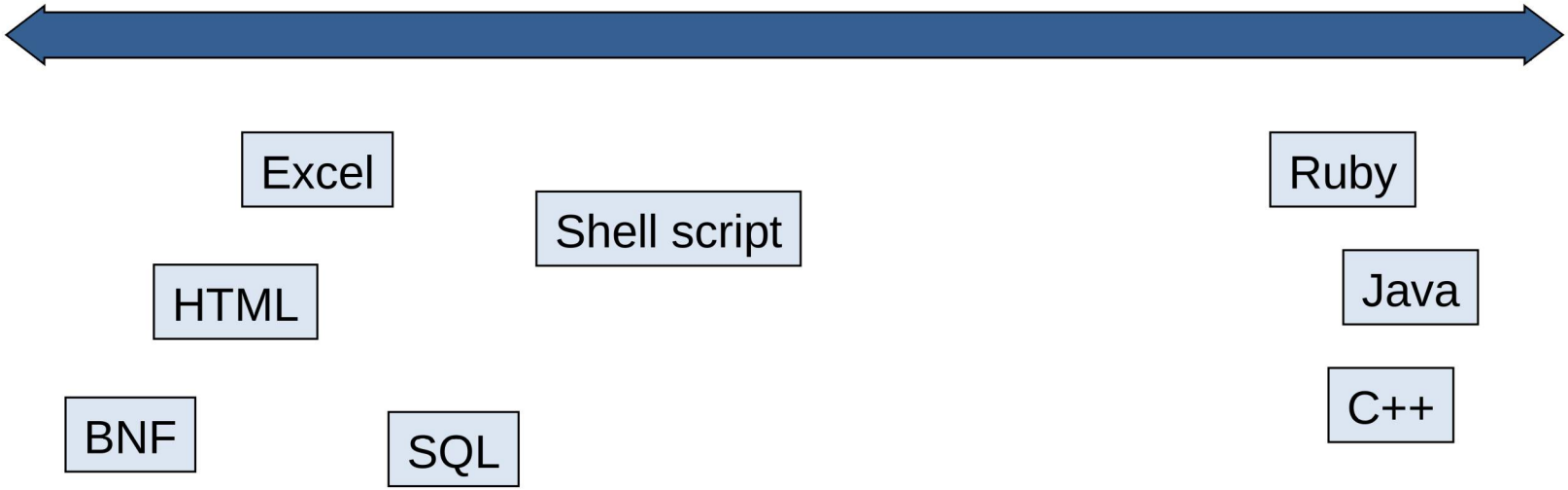
Exempel


namnlös1 - StarOffice Calc
Arkiv Redigera Visa Infoga Format Verktyg Data Fönster Hjälp
Arial
10
A9
=SUMMA(A2:A7)
A B C D E F
1 Antal Andel
2 45 29%
3 19 12%
4 23 15%
5 30 20%
6 9 6%
7 27 18%
8
9 153 100%
10
11
12
13
Tabell1 Tabell2 Tabell3
Tabell 1 / 3 Standard 100% STD *

Domänspecifik är inte entydigt

DSL
Domain Specific Language

GPL
General Purpose Language



Språken behöver inte nödvändigtvis vara körbara.

Varför vill man ha DSL?

- Bättre uttryckskraft inom ett begränsat område kan ge högre produktivitet.
  - *snabbare utveckling*
  - *kortare kod med färre fel*
- Mindre behov av att kunna allmän programmering gör att fler personer, framför allt med domänkunskap, kan vara med och utveckla ”program”.

Bättre uttryckskraft för att…

- göra saker som man gör många gånger, t.ex. testning
- definiera komplicerade datastrukturer
- lättare bearbeta datastrukturer
- konfigurera system
- konstruera grafiska gränssnitt

Kan man mäta produktivetet?

Språk	Tillämpningsområde	Nivå	Rader källkod per FP
Excel 5	Kalkyl	57	6
SQL	Databasfrågor	25	13
HTML 3.0	Webbsidor	22	15
VHDL	Design av hårdvara	17	19
Make	Bygga programvara	15	21
Java	Generellt språk	6	53
C	Generellt språk	2,5	128

Programming Languages Table (Software Productivity Research)

Kan man mäta produktivetet?

- FP ger en uppfattning om hur mycket kod som skulle behövas i ett annat språk
- Påverkan på produktivitet är inte nödvändigtvis representativ
- 30% kod 70% annat i större projekt

Implementation av DSL

- Externt DSL
  - *Ny syntax som inte är gemensam med något existerande programspråk. Kräver helt nya verktyg, men man kan skapa ett uttrycksfullt språk för det specifika problemet.*
- Internt DSL
  - *Utvidgning av ett existerande programspråk (en typ av abstraktion), med fördelen att man lätt kan bearbeta koden.*

Verktyg för körbara DSL

- Interpretator (helt ny eller utbyggd)
- Kompilator (helt ny eller utbyggd)
- Preprocessor
- API

Vi kommer att testa interna DSL (dvs som bygger på Ruby) som utnyttjar och utökar Ruby-interpretatorn.

Exempel på eget DSL

```
article 1234
name "sune"
description "book shelf"
color "red"
inventory 42
article 5678
inventory 29
name "berra"
color "brown"
description "mirror"
```

Denna textfil innehåller information om ett antal artiklar i ett lager.

Betrakta textfilen som programkod och försök skriva ett verktyg som kan tolka den. Hur då?

```
def article(n)
  # lagra artikelnummer
end

def name(the_name)
  # lagra artikelnamn
end

def description(the_text)
  # lagra beskrivning
end

def color(c)
  # lagra färg
end

def inventory(n)
  # lagra antal i lager
end
```

Detta är ett förslag på definition av metoder som kan ta emot information om artiklar.

a) Hur ska vi kunna *köra* koden som finns i filen?

b) Hur ska vi lagra informationen som läses in?

Körning av kod med eval

```
>> eval "1+3"
=> 4
>> eval "'abcd'.length"
=> 4
>> eval "def
a(n);n+2;end;a(45)"
=> 47
>> a(3)
=> 5
```

Körning av kod med eval

```
>> eval "1+3"
=> 4
>> eval "'abcd'.length"
=> 4
>> eval "def
a(n);n+2;end;a(45)"
=> 47
>> a(3)
=> 5
```

code.rb

```
puts "Hello, world!"
"All the world’s a stage".gsub(/ /, "_")
```

Körning av kod med eval

```
>> eval "1+3"
=> 4
>> eval "'abcd'.length"
=> 4
>> eval "def
a(n);n+2;end;a(45)"
=> 47
>> a(3)
=> 5
>> eval File.read("code.rb")
Hello, world!
=> "All_the_world’s_a_stage"
```

code.rb

```
puts "Hello, world!"
"All the world’s a stage".gsub(/ /, "_")
```

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.

Eval i kontexten av en klassinstans

- ***o*.instance_eval( )**
  - OOP kan vi, så låt oss lösa problemet med en klass
  - Hur kan vi skapa en instans av klassen i klassen?

```
class Warehouse < Array

  def Warehouse.load(filename)

  end

end
```

Statisk medlemsfunktion

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new

  end

end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def name(the_name)
    self.last[:name] = the_name
  end

end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def name(the_name)
    self.last[:name] = the_name
  end

  # Metoderna description, color och inventory
  # definieras på samma sätt som name.
end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def name(the_name)
    self.last[:name] = the_name
  end

  # Metoderna description, color och inventory
  # definieras på samma sätt som name.
end
```

Inläsning av data till en array av hash-tabeller.

```
>> w = Warehouse.load("lager.txt")
=> [{:article=>1234, :name=>"sune",
:description=>"book
shelf", :color=>"red",
:inventory=>42}, {:article=>5678,
:inventory=>29, :name=>"berra",
:color=>"brown", :description=>"mirror"
}]
```

Den mörka sidan av eval

- Ett gigantiskt säkerhetshål som bara väntar på att utnyttjas
- Stora möjligheter att skapa fullständigt obegriplig kod
- Svårt att felsöka

En mer generell lösning

- Vi vill kunna hantera vilka egenskaper som helst och slippa ha likadana metoder för *name*, *description*, *color*, etc.
- Vad händer om vi tar bort alla dessa metoder utom article?

```
article 1234
name "sune"
description "book shelf"
color "red"
inventory 42
article 5678
inventory 29
name "berra"
color "brown"
description "mirror"
```

En mer genrell lösning

- Metoden saknas så klart och vi kan inte anropa den...

```
irb(main):002:0>
w=Warehouse.load("lager.txt")
NoMethodError: undefined method `name'
for [{:article=>1234}]:Warehouse
        from warehouse.rb:5:in `instance_eval'
        ...
```

```
article 1234
name "sune"
description "book shelf"
color "red"
inventory 42
article 5678
inventory 29
name "berra"
color "brown"
description "mirror"
```

Anrop till metoder som saknas

1. Man kan fånga undantag, precis som i många andra språk.
2. Man kan överlagra den interna metoden **method_missing**, som anropas varje gång en metod inte finns.

Några tips angående **method_missing**:

- Se till att den kan ta godtyckligt många argument:

```
def method_missing(name, *args)
```

- Lägg under utvecklingen gärna in spårutskrifter som skriver ut argumenten, för att lättare förstå varför den anropas.

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

end
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def method_missing(name, *args)
     puts "name: #{name}, args: #{args}"
  end
end
```

Övning

- Jobba med filerna warehouse.rb och lager.txt
- Modifiera den givna koden i warehouse.rb
- Klassfunktionen ”load” ska returnera en hash med hela lagret

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def method_missing(name, *args)
    self.last[name] = args[0]
  end
end
```

```
article 1234
name "sune"
description "book shelf"
color "red"
inventory 42
article 5678
inventory 29
name "berra"
color "brown"
description "mirror"
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    # transform w to hash and return
  end

  def article(number)
    self << {:article => number}
  end

  def method_missing(name, *args)
    self.last[name] = args[0]
  end
end
```

```
article 1234
name "sune"
description "book shelf"
color "red"
inventory 42
article 5678
inventory 29
name "berra"
color "brown"
description "mirror"
```

Ett lite besvärligare dsl

1. Nu såg vi till att det i DSL bara fanns funktionsanrop och instanser (strängar, heltal)

2. Men hur hade det gått om vi inte ville skriva texten inom citattecken?

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def method_missing(name, *args)
    self.last[name] = args[0]
  end
end
```

```
article 1234
name sune
description book shelf
color red
inventory 42
article 5678
inventory 29
name berra
color brown
description mirror
```

```
class Warehouse < Array

  def Warehouse.load(filename)
    w = new
    w.instance_eval(File.read(filename))
    w
  end

  def article(number)
    self << {:article => number}
  end

  def method_missing(name, *args)
    self.last[name] = args[0]
  end
end
```

```
article 1234
name sune
description book shelf
color red
inventory 42
article 5678
inventory 29
name berra
color brown
description mirror
```

Behöver ingripa här på något sätt.

Två tekniker som gör Ruby bra för DSL

Kernel#eval
BasicObject#instance_eval
Module#class_eval

För att köra kod som finns i en sträng, en fil eller ett kodblock.

Kernel#method_missing

För att fånga upp när vi anropar en metod som inte finns.

Exempel 2 (körning)

```
zaza1 <1> cat config.rb
a 45
b 4711
c 1337
zaza1 <2> irb --simple-prompt
>> load "configuration.rb"
=> true
>> cfg=Configuration.new("config.rb")
=> #<Configuration:0x2fdda0c @c=1337, @b=4711,
@a=45>
>> cfg.a
=> 45
```

Exempel 2 (programkod)

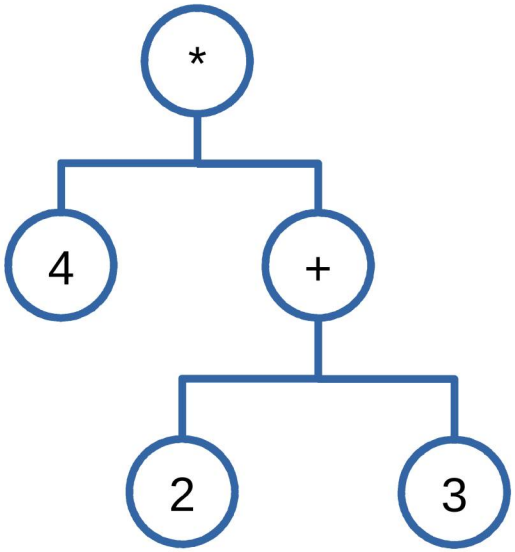
```
class Configuration

  def initialize(filename)
    instance_eval(File.read(filename))
  end

  def method_missing(name, arg)
    Configuration.class_eval(”attr_accessor :#{name}")
    instance_eval("@#{name}=#{arg}")
  end

end
```

Lösa en övning

- I filen **expr.rb** finns ett exempel på ett domänspecifikt språk för att beräkna matematiska uttryck. Varje rad består av ett variabelnamn följt av ett uttryck som detta ska bindas till.
- a 7
- b 5
- c a+3 # 10
- d c*b*2 # 100
  e d-7 # 93
- Vår uppgift är att läsa in och beräkna den här typen av uttryck med hjälp av DSL-tekniker. Exempel:
- >> Expression.calculate(”expr.rb”)
  => 93

Abstrakt syntaxträd

```
4*(2+3)
```

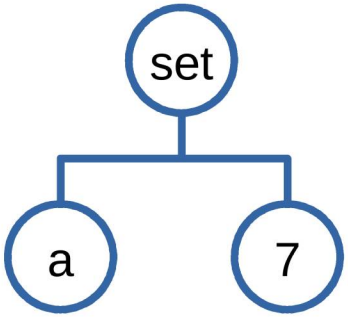


Lösa en övning (som ett träd)

Här visas ett generellt exempel på ett abstrakts sätt att representera programflödet

```
a 7
b 5
c a+3     # 10
d c*b*2   # 100
e d-7     # 93
```

Lösa en övning (bygg träd)

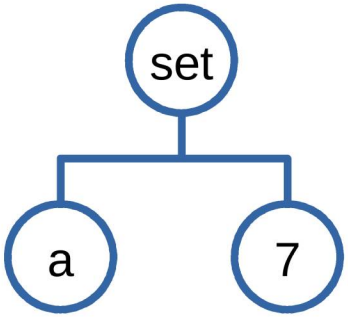



```
a 7
b 5
c a+3      # 10
d c*b*2    # 100
e d-7      # 93
```

Data

```
{

}
```

Lösa en övning (exekvera)



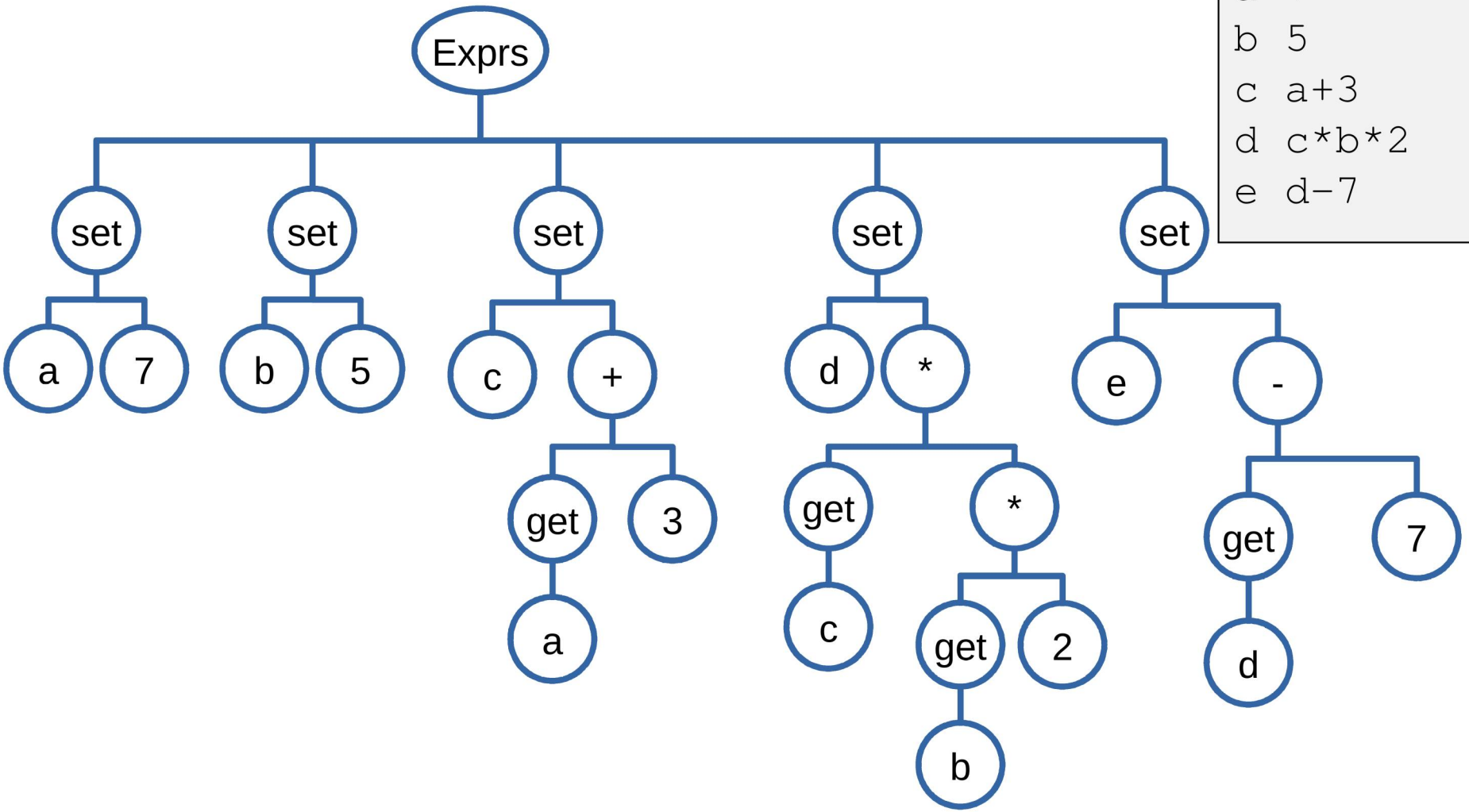

```
a 7
b 5
c a+3      # 10
d c*b*2    # 100
e d-7      # 93
```

Data

```
{
a: 7
}
```

Hela programmet som ett träd (läs)

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



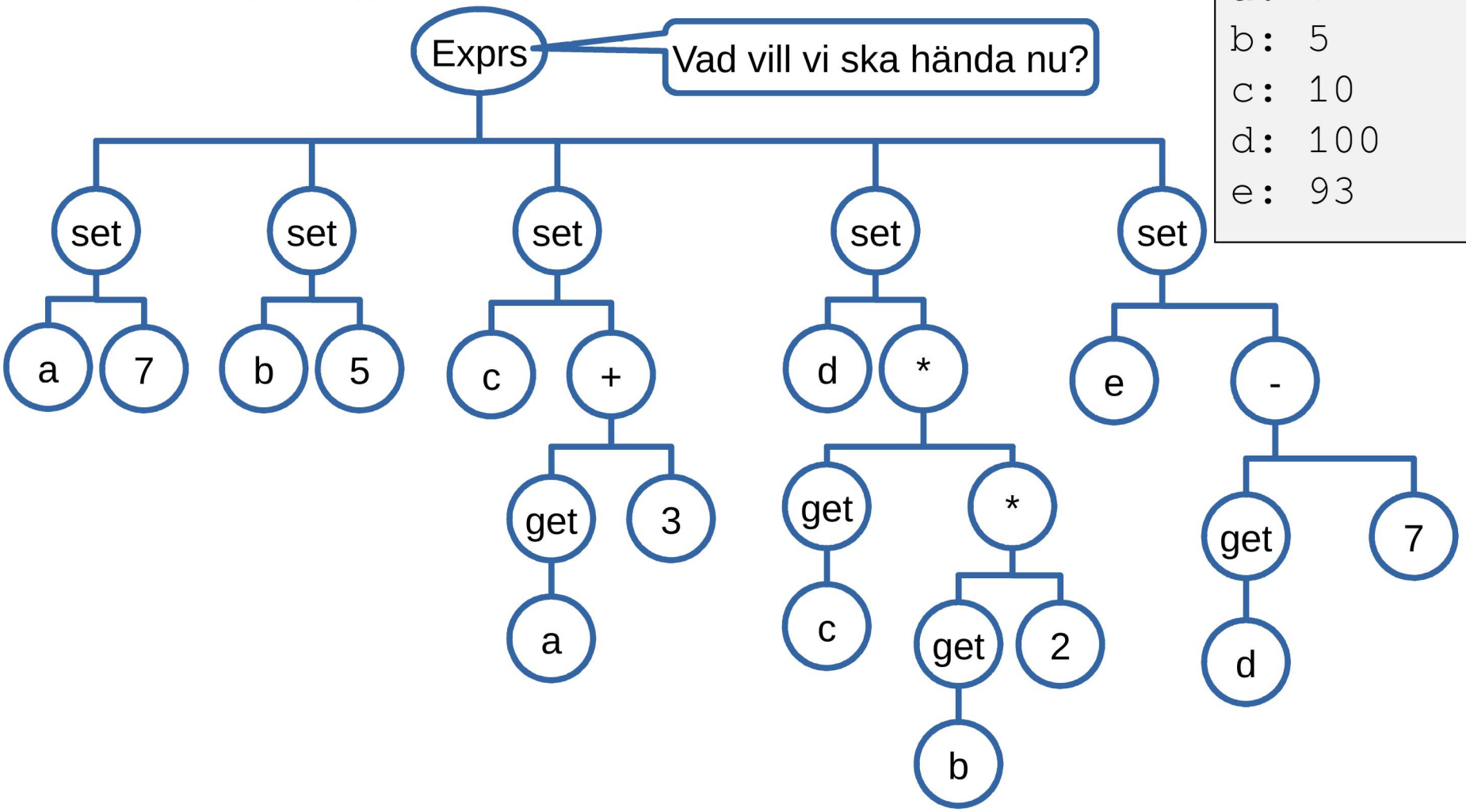
Hela programmet, exekvering

Vi måste bestämma vad som händer

Kör programmet

DATA:

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



Tillbaka till övningen

- I filen **expr.rb** finns ett exempel på ett domänspecifikt språk för att beräkna matematiska uttryck. Varje rad består av ett variabelnamn följt av ett uttryck som detta ska bindas till.
- a 7
- b 5
- c a+3 # 10
- d c*b*2 # 100
  e d-7 # 93
- Vår uppgift är att läsa filen och generera tillhörande abstrakta syntaxträd.
- Exekvera sedan trädet och se till att det returnerar det sista tilldelade värdet (e)

www.liu.se

LINKÖPINGS UNIVERSITET