## rdparse

```
require './rdparse'
MONTHS = {'januari' => 1,
'februari' => 2, 'mars' => 3,
'april' => 4,
    'maj' => 5, 'juni' => 6,
'juli' => 7, 'augusti' => 8,
'septe mber' => 9,
    'ok tober' => 10, 'november'
=> 11, 'december' => 12}
class Calendar

   def initialize
        @ca len dar Parser =
Parser.ne w("c alendar
parser ") do
           @c = Time.now
            log ger.level =
Logger ::WARN
            tok en( /'.* ?'/) {
|m| m[1..-2] } # quoted strings
            tok en( /\s+/) #
whites paces (ignored)
            tok en( /\d+/) { |m|
m.to_i } # integers
            tok en( /\w+/) { |m|
m } # alphan ume rical strings
            tok en(/./) { |m| m
} # other charac ters, indivi -
dually

            start :expr do
                mat ch( :date) {
[] }
                mat ch( :me -
eting)
           end

           rule :date do
                mat ch('-',
Integer, String, Integer, '-')
do |_, d, m, y, _|
                   @c = Time.l -
ocal(y, MONTHS [m.d ow ncase],
d)
               end
           end
           rule :meeting do
                mat ch( :time,
'-', :time, String) do | start,
```

## rdparse (cont)

```
>  end

 def read_meetings(filename)
  result = []
  content = IO.readlines(filename)
  content.each do |line|
   line.strip!
   if line.length > 0 then
    r = @calendarParser.parse(line)
    if r != [] then
     result << r
    end
   end
  end
  result
 end
end
```

## DSL

## DSL (cont)

```
>        result << pair if pair[0]==arg1
      elsif arg2 != '*' then
       result << pair if pair[1]==arg2
      end
    end
    result
  end
end
```

## Constraint network

```
_, stop, text |
                    [text,
start, stop]
               end
           end
           rule :time do
                mat ch( Int -
eger, '.', Integer) do |h, _, m|
                    Tim e.l -
oca l(@ c.year, @c.month,
@c.day, h, m)
               end
           end
       end
```

```
class TripleStore
   def initialize
       @db = {}
   end

   def Triple Sto re.l oa d(f -
ile name)
       r = new
        r.i nst anc e_e val -
(Fi le.r ea d(f ile name))
       r
   end

   def method _mi ssi ng( pro -
per ty, arg 1,arg2)
        table = @db[pr ope -
rty.to_s]
       if table then
            table << [arg1, -
arg2]
       else
            @db [pr ope -
rty.to_s] = [[arg1 ,arg2]]
       end
   end
   def find(p rop ert y,a -
rg1 ,arg2)
        table = @db[pr operty]
        result = []
        tab le.each do |pair|
           if arg1 != and
arg2 != '' then
                result << pair
if (pair[ 0]= =arg1) and
(pair[ 1]= =arg2)
            elsif arg1 != '*'
then
```

```
require "./constraint_networks"
class Squarer
    include Pretty Print
    att r_a ccessor :a, :sq_a
    att r_r eader :logger
   def initia liz e(a ,sq_a)
        @lo gge r=L ogg er.n -
ew (ST DOUT)
        @a, @sq _a= [a, sq_a]
        [a, sq_ a].each {|x|
x.add_ con str ain t(s elf)}
   end

   def to_s
        " #{a}^2 == #{sq_a }"
   end

   def new_va lue (co nne ctor)
       if a==con nector and
a.has_ value? and (not sq_a.h -
as_ value?) then
            log ger.de bug ("#
{ self} : #{sq_a} update d")
            sq_ a.a ssi gn( -
a.v alu e*a.va lue ,self)
        elsif sq_a== con nector
and sq_a.h as_ value? and (not
a.has_ value?) then
            log ger.de bug ("#
{ self} : #{a} update d")
            a.a ssi gn( Mat -
h:: sqr t(s q_a.va lue ),self)
       end
       self
   end

   # A connector lost its value,
so propagate this inform ation
to all
   # others
   def lost_v alu e(c onn ector)
        ([a ,sq _a] -[c onn -
ect or] ).each { |conne ctor|
connec tor.fo rge t_v alu -
e(self) }
   end

end
```

## Constraint network (cont)

```
> def circle_area_network
  radius=Connector.new('radius')
  radius_sq=Connector.new('radius_sq')
  area=Connector.new('area')
  pi=ConstantConnector.new('pi',Math::PI)
  Squarer.new(radius,radius_sq)
  Multiplier.new(pi,radius_sq,area)
  [radius,area]
end
$radius,$area=circle_area_network
def circle_area(radius)
  $radius.forget_value "user" if $radius.h-
as_value?
  $radius.user_assign radius
  $area.value
end
```

## Teori

PARSER: Parsern fungerar på så sätt att den först extraherar tokens ur teckenströmmen med hjälp av regex för att sedan arbetas på med hjälp av reglerna som är specificerade under initialize. De tokens som extraheras matchas enligt reglerna och ett uttryck byggs på så sätt upp från grunden. För varje regel som matchas så körs kodblocket bredvid det för att t.ex. spara bort koden i en variabel eller för att köra någon funktion med koden som inpara-meter.

DSL vs GPL: Ett DSL är ett skräddarsytt språk designat för att lösa specifika problem. Användning för ett GPL är väldigt brett och det används för att lösa otroligt många men orelaterade uppgifter. En anledning till att man kanske skulle vilja skriva ett DSL är för att simplifiera användarinmatning, möjligtvis för användare som inte är familjära med kodning. Det kan också användas för att simplifera datainput, t.ex konfiguration för ett program.

## Teori (cont)

CONTINUATION: En continuation är lite av en glorifierad GOTO. Den används för att kunna hoppa till en specifik plats någonstans i koden. Continuations sparar även 'omgiv-ningen' som den ser ut vid just det tillfället vilket gör att man kan arbeta på variabler som de såg ut då och inte nu. En anledning till att använda continuations istället för ett antal for-loopar är hur enkelt det är att ta sig tillbaka till början continuation satsen oavsett var man är i koden för tillfället, istället för att behöva breaka en massa for-satser.

LEX vs SYNT: Lexikalisk analys är när en ström tecken tolkas inte bara som enskilda tecken men de transformeras istället till tokens. t.ex. 'int 5' blir uppdelad till både 'int' och siffran 5. Syntaktisk analys är när dessa tokens används för att verifiera att de används enligt den specif-icerade grammatiken. t.ex att 'if' är följt av en jämförelse och sedan en sats.

RUBY FÖR DSL: Ruby är väldigt bra när det kommer till s.k metaprogrammering och DSL är en del av just det. En anledning till att Ruby passar för ett DSL är method_mi-ssing, alltså metoden där man man kan dynamiskt skapa en metod utan att behöva definera den tidigare. Detta leder till en väldig frihet när det kommer till syntax. instance_eval är en annan väldigt användbar del av Ruby när det kommer till DSL'er. Med instance_eval kan man skicka in ett block kod för att köras inuti en klass. En tredje fördel är metoden send där namnet på en metod skickas in som en symbol och då körs denna metod i klassen i fråga.