SWE#

Tekniska högskolan vid Linköpings universitet
Innovativ Programmering

Projektdokumentation
TDP019 – Projekt:Datorspråk
2012-05-14

Examinator
Anders Haraldsson

Marcus Hellgren, marhe414@student.liu.se
Jacob Holm, jacho391@student.liu.se

Innehållsförteckning

Inledning

Uppgiften var att skapa ett eget datorspråk som implementeras i Ruby. I den här rapporten kommer språket SWE# som vi gjort för kursen TDP019 Projekt: Datorspråk på Linköpings universitet tas upp i detalj. Dokumentet kommer innehålla hur språket är uppbyggt, vilka funktioner som finns och hur syntaxen fungerar samt hur vi har arbetat för att skapa språket.

SWE# är i grunden ett språk gjort för att kunna fungera som ett första språk för nya programmerare mer än för vana användare. Då de flesta populära språk ute på marknaden är på engelska har vi valt att utveckla ett svenskt språk som har mycket nyckelord i syntaxen i stället för symboler. Detta görs för att underlätta läsbarheten till en mer smidigare text för nya programmerare då symboler kan krångla till det och ta lång tid att lära sig i början. SWE# är ett imperativt språk med stark typning.

SWE# är ett språk som använder sig av ett verktyg som kallas rdparser för att läsa och tolka given syntax. Parsern tolkar sedan ett SWE# program till Ruby-kod, som exekveras samt översätter det och utför själva SWE#-koden i sin tur.

1. Användarhandledning

1.1 Installation

Till att börja med måste du ha Ruby installerat för att kunna köra SWE#. Vi rekommenderar att du hämtar den senaste stabila versionen från Rubys egna hemsida. Vi testade i Ruby 1.8.7 och Ubuntu 11.10.

För att kunna köra SWE# måste du ha filerna som finns i tar-filen SWE.tar. Packa upp filerna i valfri mapp.

För att du enkelt och smidigt ska kunna köra SWE# behöver du även ändra lite i din .bashrc fil. Öppna kommandotolken och öppna filen .bashrc, med förslagsvis emacs, med kommandot:

```
emacs .bashrc &
```

därefter lägger du in följande rad i filen och sparar.

```
alias SWE="ruby SWE#.rb"
```

Starta sedan om kommandotolken, navigera dig fram till mappen dit du packade upp alla filer och allt är klart för att du ska kunna börja köra SWE#.

1.2 Första programmet

Dags för det första programmet. Öppna en vanlig textfil i mappen där du placerade alla filerna. I textfilen skriver du sedan:

```
skriv "Hej Världen"
```

Spara dokumentet med valfritt namn. För att kompilera och köra koden öppnar du upp kommandotolken, ställer dig i den korrekta mappen och skriver:

```
SWE filnamn
```

Programmet du nu skrev kommer då skriva ut ”Hej Världen” till dig och du gjort ditt första SWE# program!

2. Syntax och strukturer

SWE# är inte strikt när det gäller just indentering och mellanslag, dock rekommenderar vi att man tydligt strukturerar upp sin kod för läsbarhetens skull. Det underlättar mycket när man senare ska gå igenom både sin egen och andras kod. Då hela syntaxen är skifteskänslig är det viktigt att hålla reda på vad som skrivs med versaler och vad som skrivs med gemener.

De konstruktioner som finns är tilldelning, funktioner, villkorssatser och repetitionssatser. De datatyper som finns i språket är:

Tal - Heltal

Flyt - Decimaltal

Text - Strängar, Textvärde

Sanning - Boolean, Sant eller falskt värde

2.1 Aritmetiska uttryck

Aritmetiska uttryck använder sig av de vanliga matematiska reglerna. De matematiska operatorerna som går att använda är:

*	/	+	-

alla kombinerat med parenteser.

```
5 * 5 + 3       # kommer ge värdet 28
5 + 5 * 3       # kommer ge värdet 20
5 * (5 + 3)     # kommer ge värdet 40
5 * 10 / (5 * 10)       # kommer ge värdet 1
```

Värdet av ett aritmetiskt uttryck behövs antingen sparas ner i en variabel eller skrivas ut direkt med ”skriv” funktionen om man vill se vad svaret blir. Inget synligt händer utan att man anropar ”skriv”. (Se Tilldelning)

2.2 Relationsuttryck och Jämförelseoperatorer

De operatorer som går att använda för att jämföra olika värden är:

<	<=	>	>=	!=	==

```
x < y       # x mindre än y
x <= y      # x mindre eller lika med y
x > y       # x större än y
x >= y      # x större eller lika med y
x != y      # x skillt ifrån y
x == y      # x är lika med y
```

Operatorerna kommer ge ett sanningsvärde antingen ”sant” eller ”falskt”, som går att använda i t.ex. Om-satser.

2.3 Tilldelning

Då SWE# är ett typat språk måste variabler ha en datatypsdeklaration första gången de skapas.

```
Tal y = 5 * 5 + 3        # y ges värdet 28
y = 1                    # y skrivs sedan över till 1
skriv y
```

Skriv kommer skriva ut 1 och allt kommer fungera då y typ-deklarerades när den skapades. Man behöver inte ha med typen senare om samma variabel används igen. SWE# är även skifteskänslig på variabelnamn så Y och y blir olika variabler vilket kan vara värt att ha i minnet.

Det går även att tilldela en variabel med värdet av hela uttryck med t.ex. andra variablers värden.

```
Tal x = 5
Tal y = 5
Tal z = x + 2 * y                # z ges det totala värdet 10
```

z kommer då innehålla 10. Flyt fungerar exakt likadant som vanliga tal förutom att de är decimaltal.

```
Flyt x = 2.4    # Ger x värdet 2.4
```

Datatypen Sanning har endast två olika värden, ”sant” eller ”falskt”

```
Sanning x = sant      # Sätter variabeln x till ”sant”
Sanning y = falskt    # Sätter variabeln y till ”falskt”
```

För Text gäller att strängen (den text man vill att den ska innehålla) har ” innan och efter.

```
Text x = ”Den här strängen kommer läggas i namnet x”
```

2.4 Funktioner

Funktioner behöver en definerad datatyp på både parametrarna och returvärdet av funktionen. Funktionen måste alltid ha en return definerad även om returvärdet ska vara inget(null). Man deklarerar först datatypen, sen ”funktion”, namnet på funktionen och om man vill ha parameterlista så har man det sist. Sedan skriver man alla satser man vill ska köras i den funktionen. Sist definierar man ett returvärde och avslutar med ”slutfunktion”. Returvärdet måste då matcha den datatyp man deklarerat som returvärde från funktionen.

```
Tal funktion dubblera(Tal x)
        Tal y = x * 2
        tillbaka y
slutfunktion
```

Eftersom funktionerna har ett returvärde går det att spara returvärdet i t.ex. en variabel.

```
Tal funktion dubblera(Tal x)
        Tal y = x * 2
        tillbaka y
slutfunktion
Tal z = dubblera(5)
```

Variabeln z kommer då innehålla värdet 10 då 5 tas som argument, multipliceras med 2 och sedan returneras tillbaka.

2.5 Villkorssatser - Om

Om-satser är ett sätt att testa olika typer av jämförelser och på så sätt kunna styra programmet åt olika håll och göra olika saker beroende på vad uttrycket säger. Man kan t.ex. jämföra två variabler och se vilken som är störst vilket görs i exemplet nedan. För att skriva en Om-sats börjar man med nyckelordet ”om”, sedan en jämförelse eller ett argument och sen nyckelordet ”gor”. Noterbart är att vi använder ordet ”gor” istället för ”gör” då ääö ger problem. Efter den första argumentsatsen skrivs de operationer man vill ska ske om jämförelsen/argumentet är sant. Allt ska sedan avslutas med ”slutom”.

I exemplet nedan har vi tagit med ”annars” och sedan ytterligare en operation som ska köras om den första jämförelsen/argumentet inte stämmer. Detta är valbart och man kan använda en Om-sats helt utan raderna 5 & 6 för att bara undersöka om jämförelsen/argumentet stämmer och annars inte göra någonting.

```
1.  Tal x = 2
2.  Tal y = 5
3.  om x < y gor                    # Om x är mindre än y vilket så är fallet
4.      skriv ”y är större än x” # Skriver ut att y är större än x
5.  annars
6.      skriv ”x är större y”   # Om det inte stämmer skriver den ut motsatsen
7.  slutom                          # Avslutar Om-satsen
```

Man kan göra flera jämförelser på samma gång efter varandra med ”annarsom” vilket ger flera olika tester samtidigt, exempel:

```
Tal x = 5
Tal y = 5
om x < y gor
    skriv ”svar 1”
annarsom x > y              # Om inte det första blir sant provar den nästa.
    skriv ”svar 2”
annarsom 1 < 5
    skriv ”svar 3”          # Kommer matcha på 3e satsen och skriva ”svar 3”
slutom                      # Om inget matchar avslutas Om-satsen
```

2.6 Repetitionssatser - Medans

Medans är en repetionssats som gör operationer upprepade gånger så länge ett villkor är uppfyllt. Först skrivs nyckelordet ”medans” följt av villkoret för att undersöka om medanssatsen ska köras. Därefter är det väldigt likt Om-satsen. Du har nyckelordet ”gor” följt av operationer och sist nyckelordet ”slutmedans”. Operationerna kommer köras så länge som den angivna jämförelsen/sanningsvärdet är sant och på så sätt kan du repetera igenom och köra samma operation flera gånger.

```
Tal x = 1                   # Startar på 1
medans x < 10 gor           # Så länge x är mindre än 10
        skriv x             # Skriv ut x
        x = x + 1           # Öka x med 1
slutmedans                  #Slut på medans
```

Den här koden kommer skriva ut alla siffror från 1-9 med en siffra per rad. När värdet kommer upp till 10 kommer medans brytas då jämförelsen x < 10 inte längre är sant. Något man bör tänka på vid användandet av medans är att det är lätt att råka skapa evighetsloopar om inte jämförelsen/sanningsvärdet någonstans senare ändras så att medans kan brytas.

2.7 Kommentarer

För att skriva kommentarer använder man sig av ett specialtecken, #. Allt som står efter på raden kommer att tas som en kommentar och ignoreras av programmet.

```
5 + 5   # tar 5 plus 5 och har en trevlig kommentar efter som förklarar.
```

2.8 Exempel

```
Tal funktion dubbla(Tal x)
        x = x * 2
        tillbaka x
slutfunktion

Tal x = 10
Tal y = 10
x = dubbla(y)

om x > 10 gor
        skriv "Talet ar nu storre an 10"
annars
        skriv "Talet ar mindre an 10"
slutom
```

Exemplet ovan kommer att ta ett tal och sedan dubbla det talet. Om talet är över 10 skrivs det ut annars skrivs det ut att talet är mindre än 10. Exemplet använder funktion, om-sats samt skriv.

3. Systemdokumentation

3.1 Inledning

Nedan kommer en förklaring på hur SWE# är uppbyggt och hur t.ex. scope, parsning och noder fungerar.

3.2 Lexikalisk Analys och Token

SWE# använder sig av RDparser som gör två steg. Först gör den lexikalisk analys och sedan parsning för att läsa igenom koden som skrivits i programmet och skapa passande tokens. RDparsern skapar tokens, token är en serie av tecken som plockas ut baserat på de regler vi skapat med hjälp av reguljära uttryck. Då SWE# inte är indenteringsstyrt tas alla mellanrum bort under lexikaliska analysen. Även allt efter #-tecknen tas bort då de representerar kommentarer. RDparsern skapar sedan en sekvens av tokens som är:

- Sant | Falskt
- Float
- Integer
- String
- Funktion / slutfunktion
- Keywords
- Resten

Först det plockas datatyperna ut, Sanningsvärden, Flyttal, Heltal, Strängar. För att räckvidden för variablerna skulle blir rätt kollar vi sen efter ”funktion” och ”slutfunktion”. För varje ny funktion ökar vi på räckvidden och för varje slutfunktion minskar vi ett steg. På så vis kan vi hela tiden hålla på vilken räckvidd vi är på just nu när vi parsar koden senare. Efter det tas nyckelord ut och allra sist plockas resterande tecken ut som tokens så att inget ska kunna gå igenom den lexikaliska analysen ohanterat.

3.3 Parsning

Själva parsningen är vad som skapar hela programmet och sker efter den lexikaliska analysen. Parsningen följer alla de regler som vi skapat för språket genom att följa den tillhörande grammatiken (BNF). RDparsern bygger upp ett parserträd med noder som styr flödet genom koden som vi har skrivit så att allt går som det ska.

3.4 Parseträd och Noder

När parsern går igenom koden skapas det objekt av klasser för nästan alla olika typer av konstruktioner. Dessa objekt bildar noder i parseträdet. Vi kan ta en sträng som exempel. När konstruktionen matchas skapas ett objekt av klassen text som sedan matchas vidare upp igenom parsern. Textobjektet kommer sedan läggas in i nästa objekt som matchas och skapar ett nytt klass objekt, tex tilldelning och så vidare. Toppnoden i trädet är Operationer klassen som kommer innehålla allt i det skriva programmet. Alla klasser har sedan en eval-metod som kommer evaluera noderna i parse-trädet. På så sätt kommer detta ge vår trädstruktur när varje objekt som evalueras blir en nod och sedan evaluerar vidare på nästa nod tills hela trädstrukturen har skapats. Resultatet returneras sedan upp från botten av trädet.

3.5 Räckvidd

Vi använder oss av en statisk räckvidd där all information om räckvidden finns lagrade inom en klass. När vi går in och ut genom funktioner etc och ska byta räckvidd för nästkommande rader av kod tillkallas funktioner som ändrar räckvidden fram och tillbaka beroende på vad som behövs. Räckvidden i SWE# fungerar så att du kommer åt allt inom räckvidden du är i , ex en funktion, och den globala räckvidden. Du kommer inte åt några andra variabler i en annan räckvidd om de inte ges som argument till funktionen.

3.6 Kodstandard

Vi har använt oss av en kodstandard som dels följer indentering och camelCase vilket innebär att variabelnamn inte har underscores utan ny stor bokstav för varje ”nytt” ord i variabelnamnet. De enda som börjar med stor bokstav är alla klassnamn då man enkelt ser vad som är en klass och vad som är en variabel direkt.

SWE# använder sig inte av indentering men vi rekommenderar starkt att man indenterar för att enkelt kunna läsa och strukturera sin egen kod. Dels för andras skull och dels för sin egen då bra struktur hjälper läsbarheten väldigt mycket.

4. Grammatik

4.1 Beskrivning

Grammatikregler beskrivs i BNF-grammatiken nedan. Teckenförklaring:

|	”eller”
+	”följt av”
[<x>]	”inget eller en av <x>”

4.2 BNF

```
<kör>::= <operationer>

<operation_lista> ::= <operation_lista>+<operation>
                      |<operation>

<operation> ::=       <normal_operation>

<normal_operation> ::=
                      <om_jämförelse>
                      |<medans_jamforelse>
                      |<funktion>
                      |<tilldelning>
                      |<skriv>
                      |<funk_anrop>

<om_jämförelse> ::= ”om”+<jämförelse>+”gor”+<operation_lista>+
                      [<annars_om_jämförelse>
                      |”annars”+<operation_lista
                      |<annars_om_jämförelse>+”annars”+<operation_lista>]+
                      ”slutom”

<medans_jämförelse> ::=
                      ”medans ”+<jämförelse>+”gor”+<operation_lista>+”slutmedans”

<annars_om_jämförelse> ::=
                      ”annarsom”+<jämförelse>+<operation_lista>+<annars_om_jämförelse>
                      |”annarsom”+<jämförelse>+<operation_lista>

<jämförelse> ::=      <a_uttryck>+<j_operator>+<a_uttryck>
                      |<påstående>
                      |<a_uttryck>
```

<j_operator> ::= **”<”**
|**”>”**
|**”<=”**
|**”>=”**
|**”==”**
|**”!=”**

<uttryck> ::= <jämförelse>
|<a_uttryck>

<funktion> ::= <datatyp>+**” funktion”**+<variabel>+
[(<parameterlista>]+
[<operationer_lista>]+
”tillbaka”+<a_uttryck>+**”slutfunktion”**

<parameterlista> ::= ”(”<parameterlista>”)”
|<parameterlista>+”,”+<datatyp>+<variabel>
|<datatyp>+<variabel>

<tilldelning> ::= <datatyp>+<variabel>+”=”+<uttryck>
|<variabel>+”=”+<uttryck>

<skriv> ::= **”skriv”**+<a_uttryck>

<funk_anrop> ::= <var_char>+”(”+<argumentlista>+”)”
|<var_char>+”(”+”)”

<argumentlista> ::= <argumentlista>+”,”+<a_uttryck>
|<a_uttryck>

<datatyp> ::= **”tal”**
|**”flyt”**
|**”sanning”**
|**”text”**
|**”inget”**

<a_uttryck> ::= <m_uttryck>
|<a_uttryck>+**”+”**+<m_uttryck>
|<a_uttryck>+**”-”**+<m_uttryck>

<m_uttryck> ::= <atom>
|<m_uttryck>+**”*”+**<atom>
|<m_uttryck>+**”/”**+<atom>

```
<atom>::=          ”(”<a_uttryck>”)”
                   |<påstående>
                   |<funk_anrop>
                   |<variabel>
                   |<flyt>
                   |<tal>
                   |<text>

<påstående> ::=    ”sant”
                   |”falskt”

<variabel> ::=     <var_char>+<variabel>
                   |<var_char>

<var_char> ::=     /[A-Öa-ö_]/

<text> ::=         /^”.+”/

<flyt> ::=         <tal> ”.” <tal>
                   |”-”+<flyt>

<tal> ::=          <siffra>
                   |<tal>+<siffra>
                   |”-”+<tal>

<siffra> ::=       /[0-9]/
```

5. Erfarenheter och reflektioner

Vi har lärt oss väldigt mycket om språk under projektets gång. Inte endast om vårt egna språk och Ruby utan även om hur språk är uppbyggda i allmänhet. Att lära sig bakomliggande konstruktioner och hur man kan bygga upp språk på olika sätt har varit väldigt lärorikt. Vi hade förväntat oss väldigt mycket jobb och problem, speciellt i början av projektet. Något som dock gick mycket lättare än vi förväntade oss var aritmetiska uttryck. Vi börjar med att testa en av de första implementationer av språket och det fungerade nästan direkt med mindre modifieringar. Det var inget vi hade räknat med alls då det kändes mer komplicerat än det var. Planering för hur språket skulle se ut tog ganska mycket av vår tid innan vi började implementera. Tack vare det så behövde vi inte ändra och tänka om lika mycket utan kunde mer arbeta på utifrån det vi bestämt innan.

Det absolut svåraste vi implementerade var Scope-hanteringen. Vi började göra ett scope i matchningen som inte alls fungerade. Lösning vi tillslut arbetade fram var att göra scope-hanteringen redan vid lexikaliska analysen. Där använder vi nyckelord för att kunna skapa nya scopes som hanteras med en klass som också heter Scope och med hjälp av olika structs.

Jämför man den första grammatiken med vår färdiga grammatik är det inte så stora skillnader alls faktiskt. Vi har lyckats bra med att följa grammatiken och de ändringar vi gjort är egentligen bara ”komprimeringar” då vi upptäckt att t.ex. vissa saker kunde läggas ihop. Vi hade några saker som vi inte hade tid att implementera som t.ex. sumla och rekursion. Sumla skulle vara en funktion som ger en slumpad siffra mellan ett givet intervall. När vi hittade en bug i språket och löste den gav det upphov till mycket nya problem som tog upp väldigt mycket tid. Därför fick vi ingen tid över till att kunna göra klart sumla och rekursion som är två saker vi gärna hade velat ha med i språket.

Målet med vårat språk var att skapa ett enkelt, lätt nybörjarspråk på svenska. Vi tycker att vi lyckats bra med det målet. Vårt språk ger en enkel start för den som vill börja programmera och vill starta på svenska. Språket har många grundkonstruktioner som används i flertalet av de populärare språken vilket ger en bra start. Allt som allt är vi nöjda med hur resultatet av språket blev även fast vi saknade två konstruktioner (sumla och rekursion) som vi ville ha med. Detta gör dock egentligen ingenting då det inte ändrar något för nybörjare då speciellt rekursion är en lite mer avancerad konstruktion.

6. Bilagor

De filer som används inom språket SWE#. Filerna är uppdelade i 4 olika filer. Se bilagor nedan.

Bilaga 1 SWE#.rb

```
require 'rules'

file = ARGV[0]

if file != nil
  swe = Rules.new(file)
  swe.start
else
  puts "Filnamn saknas!"
end
```

Bilaga 2 classes.rb

```
class Operationer
  def initialize(operation)
    @operationList = []
    @operationList << operation
  end

  def add(operation)
    @operationList << operation
  end

  def setScope(scope)
    @operationList.each { |operation| operation.setScope(scope)}
  end

  def eval
    @operationList.each { |operation|
      operation.eval
    }
    nil
  end

  #returns declarations of variables
  def getNewVariables
    scope = Hash.new(Struct::Variable.new(Datatyp.new(Inget), Inget.new))

    @operationList.each { |operation|
        if operation.getVariable.class == Variable
          datatype,name,expr = operation.getVariable.to_a
          scope[name] = Struct::Variable.new(datatype, expr)
      end
    }
    scope
  end

  #returns updates to already declared variables
  def getVariableUpdates
    updates = {}
    @operationList.each {|operation|
      if operation.getVariable.class == VariableUpdate
        var = operation.getVariable
        updates[var.getName] = var.getValue
      end
    }
    updates
  end
end
#### End of class Operationer

class Operation
  def initialize(stmt)
    #Seperate Variables as they are handled in a different way
    if stmt.class == Variable or stmt.class == VariableUpdate
      @stmt = Inget.new
      @var = stmt
    else
      @stmt = stmt
      @var = Inget.new
    end
  end

  def setScope(scope)
    @stmt.setScope(scope)
    @var.setScope(scope)
  end

  def eval
    return @stmt.eval()
```

```
  end

  def getVariable
    return @var
  end
end
#### End of class Operation

#The print class
class Skriv
  def initialize(expr)
    @expr = expr
    @scope = 0
  end

  def setVariableList(variables)
    @@variables = variables
  end

  def setScope(newScope)
    @scope = newScope
    if @expr.class == Uttryck
      @expr.setScope(newScope)
    end
  end

  def eval
    expr = @expr
    if @expr.class == String #indicates a Variable call
       expr = @@variables.get(@expr, @scope)
    end
    if @expr.class == FunktionAnrop #need to setScope
      expr.setScope(@scope)
    end
    puts expr.eval
  end
end
#### End of class Skriv

class Text
  attr_accessor :datatype
  def initialize(value)
    @value = value
    @datatype = Datatyp.new(self.class)
  end
  def eval
    return @value
  end
end
#### End of class Text

class Uttryck
  attr_accessor :datatype
  def initialize(exprLh,operator,exprRh)
    @exprLh = exprLh
    @operator = operator
    @exprRh = exprRh

    @scope = 0
    @datatype = Datatyp.new(Tal)
    @test = true
  end

  def setVariableList(variableList)
    @@variables = variableList
  end

  def setScope(scope)
    if @exprLh.class == Uttryck
      @exprLh.setScope(scope)
    end
```

```
    if @exprRh.class == Uttryck
      @exprRh.setScope(scope)
    end

    @scope = scope
  end

  def getTal
    #temp variables is used as we dont want to replace the expr with the variable-expr, only eval it
for this instance of the class
    if @exprLh.class == String
      tempExprLh = @@variables.get(@exprLh, @scope)
    else
      tempExprLh = @exprLh
    end

    if @exprRh.class == String
      tempExprRh = @@variables.get(@exprRh, @scope)
    else
      tempExprRh = @exprRh
    end

    if @operator == "*"
      result = Tal.new(tempExprLh.eval * tempExprRh.eval)
    elsif @operator == "/"
      result = Tal.new(tempExprLh.eval / tempExprRh.eval)
    elsif @operator == "-"
      result = Tal.new(tempExprLh.eval - tempExprRh.eval)
    elsif @operator == "+"
      result = Tal.new(tempExprLh.eval + tempExprRh.eval)
    end

    result
  end

  def eval
    getTal.eval
  end

end
#### End of class Uttryck

class Tal
  attr_accessor :datatype

  def initialize(value)
    @value = value
    @datatype = Datatyp.new(self.class)
  end

  def eval
    @value
  end
end
#### End of class Tal

class Sanning
  attr_accessor :datatype

  def initialize(value)
    @bool = value
    @datatype = Datatyp.new(self.class)
  end

  def eval
    if @bool == 1
      return "sant"
    else
      return "falskt"
    end
  end
```

```
end
#### End of class Sanning

#The nil value
class Inget
  attr_accessor :datatype

  def initialize
    @datatype = Datatyp.new(self.class)
  end

  def setScope(scope)
    nil
  end

  def eval
    return "inget"
  end
end
#### End of class Inget

class Variable
  def initialize(datatype, name, expr)
    @datatype = datatype
    @name = name
    @expr = expr
    @scope = 0
  end

  def setScope(scope)
    @scope = scope
    if @expr.class == Uttryck
      @expr.setScope(scope)
    end
  end

  def to_a
    return [@datatype, @name, @expr]
  end

  def eval
    Inget.new
  end
end
#### End of class Variable

#Used for updating already declared variables
class VariableUpdate
  def initialize(name, expr)
    @name = name
    @expr = expr
  end

  def setScope(scope)
    if @expr.class == Uttryck
      @expr.setScope(scope)
    end
  end

  def getValue
    return @expr
  end

  def getName
    return @name
  end
end
#### End of class VariableUpdate

#Handles all the variables
class VariableList
```

```
  attr_accessor :currentScope, :prevScope

  def initialize
    #Create struct
    Struct.new("Variable", :datatype, :expr)

    #Set standard value to hash
    globalScope = Hash.new(Struct::Variable.new(Datatyp.new(Inget), Inget.new))

    @scopes = {}
    @scopes[0] = globalScope
  end

  #Adds a Variable of the given data to given scope
  def add(datatype, name, expr, scope)
    @scopes[scope][name] = Struct::Variable.new(datatype, expr)
  end

  def updateVariable(name, expr, scope)
    #Does the variable exist on given scope?
    if @scopes[scope][name] != nil
      if expr.class == String
        expr = get(expr, scope)
      end

      if @scopes[scope][name].datatype.isValid(expr)
        if expr.class == Uttryck
          expr = expr.getTal
        end
        @scopes[scope][name].expr = expr

      else
        puts "GÃ¥r ej tilldela variabel #{name} datatyp #{expr.class} dÃ¥ den Ã¤r av datatyp
#{@scopes[scope][name].expr.class}"
      end

    #Does the variable exist on global scope?
    elsif @scopes[0][name] != nil
      if @scopes[0][name].datatype.isValid(expr)
        @scopes[0][name].expr = expr
      else

        puts "GÃ¥r ej tilldela variabel #{name} datatyp #{expr.class} dÃ¥ den Ã¤r av datatyp
#{@scopes[0][name].expr.class}"
      end

    else
      puts "Variabel #{name} Ã¤r ej deklarerad, kan inte tilldela vÃ¤rde"
    end
  end

  def get(name, scope)
    #variable on given-scope/global-scope
    if @scopes[scope][name] != nil
      return @scopes[scope][name].expr
    else
      return @scopes[0][name].expr
    end
  end

  def addScope(scope, variables)
    @scopes[scope] = Hash.new(Struct::Variable.new(Datatyp.new(Inget), Inget.new))
    @scopes[scope] = variables
  end
end
#### End of class VariableList

class Datatyp
  attr_accessor :classType

  def initialize(classType)
```

```
    @classType = classType
  end

  def isValid(x)
    if x.class == FunktionAnrop
       return true
    end

    x.datatype.classType == @classType
  end
end
#### End of class Datatyp

class Funktion
  def initialize(name, operationList, datatype, returnExpr, parameterList, scope, variableUpdates)
    @name = name
    @operationList = operationList
    @datatype = datatype
    @returnExpr = returnExpr
    @parameterList = parameterList
    @scope = scope

    if variableUpdates.class != Inget
      @variableUpdates = variableUpdates
    else
      @variableUpdates = {}
    end
  end

  def getName
    @name
  end

  def getScope
    @scope
  end

  def getParameterList
    @parameterList
  end

  def getVariableUpdates
    @variableUpdates
  end

  def eval
    @operationList.eval
    if @returnExpr.class == Struct::Variable
      @returnExpr.expr.eval
    else
      @returnExpr.eval
    end
  end
end
#### End of class Funktion

class FunktionAnrop

  def initialize(funcName,argList)
    @funcName = funcName
    @scope = 0
    @arguments = argList.getArguments
  end

  def setVariableList(variables)
    @@variables = variables
  end

  def setScope(scope)
    @scope = scope
  end
```

```
  def eval
    func = @@variables.get(@funcName, @scope)

    #Update parameters with the arguments
    if func.getParameterList.class != Inget
      func.getParameterList.getNumbers.each {|name,number|
        value = @arguments[number]
        datatype = func.getParameterList.getParameters[name].datatype
        @@variables.updateVariable(name, value, func.getScope)
      }
    end

    #Make sure all variables is updated
    func.getVariableUpdates.each{|name,expr|
      @@variables.updateVariable(name,expr,func.getScope)
    }
    return func.eval
  end
end
#### End of class FunktionAnrop

class ParameterLista
  def initialize(datatype, name)
    @variables = {}
    @numbers = {}
    @counter = 0

    @numbers[name] = @counter
    @variables[name] = Struct::Variable.new(datatype, Inget.new)
  end

  #Each parameter gets paired with a number to be able to match the arguments later
  def add(datatype, name)
    @counter += 1
    @numbers[name] = @counter
    @variables[name] = Struct::Variable.new(datatype, Inget.new)
  end

  def getNumbers
    @numbers
  end

  def getParameters
    @variables
  end
end
#### End of class ParameterLista

class ArgumentLista
  def initialize(arg)
    @counter = 0
    @arguments = {}

    if arg.class != Inget
      @arguments[@counter] = arg
    end
  end

  #Each argument gets paired with a number to be able to match the parameters later
  def add(arg)
    @counter += 1
    @arguments[@counter] = arg
  end

  def getArguments
    return @arguments
  end
end
#### End of class ArgumentLista
```

```ruby
class Scope
  attr_accessor :currentScope, :prevScope

  def initialize
    @scopeList = {}
    @scopeList[0] = 0 #pair the scopes with its previous scope
    @scope = 0
    @prevScope = 0
    @highestScope = 0
  end

  #returns a unique scope
  def getNewScope
    @prevscope = @scope
    @highestScope += 1
    @scope = @highestScope
    @scopeList[@scope] = @prevScope

    @scope
  end

  def closeScope
    @scope = @prevscope
    @prevscope = @scopeList[@prevScope]
  end
end
#### End of class Scope

class Jamforelse
  attr_accessor :datatype

  def initialize(exprLh, rOperator, exprRh)
    @exprLh = exprLh
    @rOperator = rOperator
    @exprRh = exprRh
    @scope = 0

    #Jamforelse is a valid value for Sanning datatype
    @datatype = Datatyp.new(Sanning)
  end

  def setVariableList(variableList)
    @@variables = variableList
  end

  def setScope(scope)
    @scope = scope
  end

  def eval
    value = false
    tempExprLh = @exprLh
    tempExprRh = @exprRh

    #temp variables is used as we dont want to replace the expr with the variable-expr, only eval it
for this instance of the class
    if @exprLh.class == String
      tempExprLh = @@variables.get(@exprLh, @scope)
    end
    if @exprRh.class == String
      tempExprRh = @@variables.get(@exprRh, @scope)
    end

    if @rOperator == "<"
      if tempExprLh.eval < tempExprRh.eval
        value = true
      end
    elsif @rOperator == "<="
      if tempExprLh.eval <= tempExprRh.eval
        value = true
      end
```

```
    elsif @rOperator == ">"
      if tempExprLh.eval > tempExprRh.eval
        value = true
      end
    elsif @rOperator == ">="
      if tempExprLh.eval >=tempExprRh.eval
        value = true
      end
    elsif @rOperator == "=="
      if tempExprLh.eval == tempExprRh.eval
        value = true
      end
    elsif @rOperator == "!="
      if tempExprLh.eval != tempExprRh.eval
        value = true
      end
    end
    return value
  end
end
#### End of class Jamforelse

class OmJamforelse
  def initialize(jamforelse,operationer)
    @om = [jamforelse, operationer]
    @annarsOm = Inget.new
    @annars = Inget.new
    @scope = 0
  end

  def setAnnars(operationList)
    @annars = operationList
  end

  def setAnnarsOm(operationList)
    @annarsOm = operationList
  end

  def setScope(scope)
    @scope = scope
    @om[0].setScope(scope)

    if @annarsOm.class != Inget
      @annarsOm.each {|jamforelse,_| jamforelse.setScope(scope) }
    end
    if @annars.class != Inget
      @annars.setScope(scope)
    end
  end

  def setVariableList(variables)
    @@variables = variables
  end

  def eval
    #Variable-call
    if @om[0].class == String
      @om[0] = @@variables.get(@om[0], @scope)
    end

    #Update the variables and do eval
    allFalse = true
    #if
    if @om[0].eval == true
      @om[1].getVariableUpdates.each {|name,expr|
        @@variables.updateVariable(name,expr,@scope)
      }
      allFalse = false
      @om[1].eval

    #elseif
```

```
    elsif @annarsOm.class != Inget
      @annarsOm.reverse.each {|jamforelse,operationer|
        if jamforelse.eval == true
          operationer.getVariableUpdates.each {|name,expr|
            @@variables.updateVariable(name,expr,@scope)
          }

          allFalse = false
          operationer.eval
        end

        if !allFalse
          break
        end
      }
    end

    #else
    if allFalse == true
      if @annars.class != Inget
        @annars.getVariableUpdates.each {|name,expr|
          @@variables.updateVariable(name,expr,@scope)
        }
        @annars.eval
      end
    end
  end
end
#### End of class OmJamforelse

class MedansJamforelse
  def initialize(jamforelse,operationer)
    @medans = [jamforelse, operationer]
    @scope = 0
  end

  def setScope(x)
    @scope = x
    @medans[0].setScope(x)
    @medans[1].setScope(x)
  end

  def setVariableList(x)
    @@variables = x
  end

  def eval
    #Update variables and do evals
    while @medans[0].eval
       @medans[1].getVariableUpdates.each {|name,expr|
        @@variables.updateVariable(name,expr,@scope)
      }
      @medans[1].eval
    end
  end
end
#### End of class MedansJamforelse
```

Bilaga 3 rules.rb

```
require 'rdparse'
require 'classes'


class Rules

  def initialize(file)
    @file = file
    @ruleparser = Parser.new("rules") do

      #VariableList and scope init
      @variables = VariableList.new
      @currentScope = 0
      @scope = Scope.new

      #First init of classes in need of the variableList as static
      dummySkriv = Skriv.new("dummy")
      dummySkriv.setVariableList(@variables)

      dummyFunktionAnrop = FunktionAnrop.new("dummy", ArgumentLista.new(Inget.new))
      dummyFunktionAnrop.setVariableList(@variables)

      dummyUttryck = Uttryck.new(Inget.new,Inget.new,Inget.new)
      dummyUttryck.setVariableList(@variables)

      dummyJamforelse = Jamforelse.new(Inget.new, Inget.new,Inget.new)
      dummyJamforelse.setVariableList(@variables)

      dummyOmJamforelse = OmJamforelse.new(Inget.new,Inget.new)
      dummyOmJamforelse.setVariableList(@variables)

      dummyMedansJamforelse = MedansJamforelse.new(Inget.new,Inget.new)
      dummyMedansJamforelse.setVariableList(@variables)

      #Structs
      Struct.new("Str", :value)
      Struct.new("Function", :scope, :prevScope)


      ##Tokens
      token(/\s/) #Remove whitespaces
      token(/#.+/) #Remove comments
      token(/falskt/) {Sanning.new(0)}
      token(/sant/) {Sanning.new(1)}
      token(/-?\d+\.\d+/) {|x| x.to_f} #Match Float
      token(/-?\d+/) {|x| x.to_i} #Match Integer

      #Match String
      token(/\".+\"/) {|x|
        Struct::Str.new(x)
      }

      #Match function/slutfunktion for scoping
      token(/funktion/) {
        @currentScope = @scope.getNewScope
        Struct::Function.new(@currentScope)
      }
      token(/slutfunktion/) {|x| @currentScope = @scope.closeScope
        x}

      token(/[\w]+/) {|x| x} #Match keywords
      token(/./) {|x| x} #Match rest
      ##end Tokens

      start :kor do
        match(:operation_lista) {|operationList|
```

```
    #Variables declaration to global scope
    @variables.addScope(0, operationList.getNewVariables)

    #Variable updates to global scope
    operationList.getVariableUpdates.each{|varName,expr|
      @variables.updateVariable(varName,expr,0)
    }
    operationList.eval()
  }

end

rule :operation_lista do
  match(:operation_lista, :operation) {|operationList,operation|
    operationList.add(operation)
    operationList
  }

  match(:operation) {|operation| Operationer.new(operation)}
end

rule :operation do
  match(:normal_operation) {|operation| Operation.new(operation)}
end

rule :normal_operation do
  match(:om_jamforelse)
  match(:medans_jamforelse)
  match(:funktion)
  match(:tilldelning)
  match(:skriv)
  match(:funk_anrop)
end

rule :funktion do
  #Without parameterlist
  match(:datatyp, Struct::Function, String, :operation_lista,
   "tillbaka", :a_uttryck, "slutfunktion") {
  |datatype,funcStruct,name,operationList,_,returnExpr,_|

    operationList.setScope(funcStruct.scope)
    #Variables is collected
    @localVariables = operationList.getNewVariables

    #Variable call
    if returnExpr.class == String
      returnExpr = @localVariables[returnExpr]
    end

    if datatype.isValid(returnExpr)
      @variables.addScope(funcStruct.scope, @localVariables)
      func = Funktion.new(name,operationList,datatype,
                          returnExpr,Inget.new,
                          funcStruct.scope,Inget.new)
    else
      func = Inget.new
      puts "Tillbaka vÃ¤rdet stÃ¤mmer inte Ã¶verens med funktions-definieringen"
    end

    #The function will be added the the variableList
    Variable.new(datatype,name,func)
  }

  #Function without operations just return value
  match(:datatyp, Struct::Function, String, "tillbaka", :a_uttryck, "slutfunktion") {
  |datatype,funcStruct,name,_,returnExpr,_|

    if datatype.isValid(returnExpr)
      func = Funktion.new(name,Inget.new,datatype,
                          returnExpr,Inget.new,
                          funcStruct.scope,Inget.new)
```

```
    else
      func = Inget.new
      puts "Tillbaka vÃ¤rdet stÃ¤mmer inte Ã¶verens med funktions-definieringen"
    end

    #The function will be added the the variableList
    Variable.new(datatype,name,func)
  }

  #With parameterlist and statements
  match(:datatyp, Struct::Function, String,  :parameterlista, :operation_lista, "tillbaka",
 :a_uttryck, "slutfunktion") {
 |datatype,funcStruct,name,parameters,operationList,_,returnExpr,_|

    operationList.setScope(funcStruct.scope)

    #Variable declarations, parameter variables and variable updates is collected
    @localVariables = parameters.getParameters
    @localVariables= @localVariables.merge(operationList.getNewVariables)
    @localVariableUpdates = operationList.getVariableUpdates

    #Variable call
    if returnExpr.class == String
      returnExpr = @localVariables[returnExpr]
    end

    if datatype.isValid(returnExpr)
      @variables.addScope(funcStruct.scope, @localVariables)
      func = Funktion.new(name,operationList,datatype,
                          returnExpr,parameters,
                          funcStruct.scope, @localVariableUpdates)
    else
      func = Inget.new
      puts "Tillbaka vÃ¤rdet stÃ¤mmer inte Ã¶verens med funktions-definieringen"
    end

    #The function will be added the the variableList
    Variable.new(datatype,name,func)
  }
end

rule :parameterlista do
  match("(", :parameterlista, ")") {|_,parameterList,_| parameterList}

  #Match more parameters
  match(:parameterlista, ",", :datatyp, String) {|parameterList,_,datatype,name|
    parameterList.add(datatype,name)
    parameterList
  }

  #Match first parameter
  match(:datatyp, String) {|datatype,name| ParameterLista.new(datatype,name)}
end

rule :tilldelning do
  match(:datatyp, String, "=", :uttryck) {|datatype,name,_,expr|
    objReturn = Inget.new

    if datatype.isValid(expr)
      objReturn = Variable.new(datatype,name,expr)
    else
      puts "Datatypen Ã¤r av #{datatype.classType} angivet vÃ¤rde uppfyller inte dessa krav"
    end
    objReturn
  }

  match(:datatyp, String) {|datatype,name|
    Variable.new(datatype,name,Inget.new)}

  match(String, "=", :uttryck) {|name,_,expr| VariableUpdate.new(name,expr)}
end
```

```
rule :datatyp do
  match("Tal") {Datatyp.new(Tal)}
  match("Flyt") {Datatyp.new(Tal)}
  match("Sanning") {Datatyp.new(Sanning)}
  match("Text") {Datatyp.new(Text)}
  match(:inget) {Datatyp.new(Inget)}
end

rule :inget do
  match("Inget") {Inget.new}
end

rule :skriv do
  match("skriv",:a_uttryck) {|_,aExpr| Skriv.new(aExpr)}
end

rule :medans_jamforelse do
  match("medans", :jamforelse, "gor", :operation_lista, "slutmedans") {
    |_,relExpr,_,operationList,_|MedansJamforelse.new(relExpr,operationList)}
end

rule :om_jamforelse do
  match("om",:jamforelse, "gor", :operation_lista, "slutom") {
    |_,relExpr,_,operationList,_| OmJamforelse.new(relExpr,operationList)}

  match("om",:jamforelse, "gor", :operation_lista,
        "annars", :operation_lista, "slutom") {
    |_,relExpr,_,operationList,_,elseOperationList,_|

    x = OmJamforelse.new(relExpr,operationList)
    x.setAnnars(elseOperationList)
    x
  }

  match("om",:jamforelse, "gor", :operation_lista,
        :annars_om_jamforelse, "slutom") {
    |_,relExpr,_,operationList,elseIfOperation,_|

    x = OmJamforelse.new(relExpr,operationList)
    x.setAnnarsOm(elseIfOperation)
    x
  }

  match("om",:jamforelse, "gor", :operation_lista,
        :annars_om_jamforelse,"annars", :operation_lista, "slutom") {
    |_,relExpr,_,operationList,elseIfOperationList,_,elseOperationList,_|

    x = OmJamforelse.new(relExpr,operationList)
    x.setAnnarsOm(elseIfOperationList)
    x.setAnnars(elseOperationList)
    x
  }
end

rule :annars_om_jamforelse do
  match("annarsom", :jamforelse, :operation_lista, :annars_om_jamforelse) {
    |_,relExpr,operationList,elseIfOperationList|

    elseIfOperationList << [relExpr,operationList]
    elseIfOperationList
  }

  match("annarsom", :jamforelse, :operation_lista) {
    |_,relExpr,operationList| [[relExpr,operationList]]}
end

rule :jamforelse do
  #normal relation expression
  match(:a_uttryck, :j_operator, :a_uttryck) {|aExprLh,relOperator,aExprRh|
    Jamforelse.new(aExprLh,relOperator,aExprRh)}
```

```
  #Jamforelse with only "sant" and "falskt" as expr
  match(:pastaende) {|bool|
    objReturn = Jamforelse.new(Inget.new,"!=",Inget.new)
    if bool.eval == "sant"
      objReturn = Jamforelse.new(Inget.new,"==",Inget.new)
    end
    objReturn
  }

  #Need to be able to nestle and use variables
  match(:a_uttryck)
end

rule :j_operator do
  match("<")
  match("<=")
  match(">")
  match(">=")
  match("==")
  match("!=")
end

rule :uttryck do
  match(:jamforelse)
  match(:a_uttryck)
end

rule :a_uttryck do
  match(:a_uttryck, '+', :m_uttryck) {|aExprLh,aOperator,aExprRh|
    Uttryck.new(aExprLh,aOperator,aExprRh)}

  match(:a_uttryck, '-', :m_uttryck) {|aExprLh,aOperator,aExprRh|
    Uttryck.new(aExprLh,aOperator,aExprRh)}

  match(:m_uttryck)
end

rule :m_uttryck do
  match(:m_uttryck, '*', :atom) {|aExpr,aOperator,atom|
    Uttryck.new(aExpr,aOperator,atom)}

  match(:m_uttryck, '/', :atom) {|aExpr,aOperator,atom|
    Uttryck.new(aExpr,aOperator,atom)}

  match(:atom)
end

rule :atom do
  match("(", :a_uttryck, ")") {|_,aExpr,_| aExpr}
  match(:pastaende)
  match(:inget)
  match(:funk_anrop)
  match(:variabel)
  match(:flyt)
  match(:tal)
  match(:text)
end

rule :variabel do
  match(String)
end

rule :text do
  match(Struct::Str) {|string| Text.new(string.value)}
end

rule :flyt do
  match(Float) {|float| Tal.new(float)}
end
```

```
      rule :tal do
        match(Integer) {|integer| Tal.new(integer)}
      end

      rule :pastaende do
        match(Sanning)
      end

      rule :funk_anrop do
        match(String,"(",:argumentlista,")") {|name,_,arguments,_|
          FunktionAnrop.new(name, arguments)}

        match(String,"(",")") {|name,_,_|
          FunktionAnrop.new(name, ArgumentLista.new(Inget.new))}
      end

      rule :argumentlista do
        match(:argumentlista, "," ,:a_uttryck) {|arguments,_,aExpr|
          arguments.add(aExpr)
          a
        }

        match(:a_uttryck) {|aExpr| argument = ArgumentLista.new(aExpr)}
      end

    end
  end

  def start
    compile = File.read(@file)
    @ruleparser.logger.level =  Logger::WARN
    puts "=> #{@ruleparser.parse compile}"
    nil
  end

end
```

Bilaga 4 rdparse.rb

```
#!/usr/bin/env ruby

# 2010-02-11 New version of this file for the 2010 instance of TDP007
#   which handles false return values during parsing, and has an easy way
#   of turning on and off debug messages.

require 'logger'

class Rule

  # A rule is created through the rule method of the Parser class, like this:
  #   rule :term do
  #     match(:term, '*', :dice) {|a, _, b| a * b }
  #     match(:term, '/', :dice) {|a, _, b| a / b }
  #     match(:dice)
  #   end

  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    @logger = parser.logger
    # The name of the expressions this rule matches
    @name = name
    # We need the parser to recursively parse sub-expressions occurring
    # within the pattern of the match objects associated with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #   match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.
  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
    if pattern[0] == @name
      pattern.shift
      @lrmatches << match
    else
      @matches << match
    end
  end

  def parse
    # Try non-left-recursive matches first, to avoid infinite recursion
    match_result = try_matches(@matches)
    return nil if match_result.nil?
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result if result.nil?
      match_result = result
    end
  end

  private

  # Try out all matching patterns of this rule
  def try_matches(matches, pre_result = nil)
    match_result = nil
    # Begin at the current position in the input string of the parser
    start = @parser.pos
    matches.each do |match|
      # pre_result is a previously available result from evaluating expressions
      result = pre_result ? [pre_result] : []
```

```
      # We iterate through the parts of the pattern, which may be e.g.
      #   [:expr,'*',:term]
      match.pattern.each_with_index do |token,index|

        # If this "token" is a compound term, add the result of
        # parsing it to the "result" array
        if @parser.rules[token]
          result << @parser.rules[token].parse
          if result.last.nil?
            result = nil
            break
          end
          @logger.debug("Matched '#{@name} = #{match.pattern[index..-1].inspect}'")
        else
          # Otherwise, we consume the token as part of applying this rule
          nt = @parser.expect(token)
          if nt
            result << nt
            if @lrmatches.include?(match.pattern) then
              pattern = [@name]+match.pattern
            else
              pattern = match.pattern
            end
            @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #{pattern.inspect}'")
          else
            result = nil
            break
          end
        end
      end
      if result
        if match.block
          match_result = match.block.call(*result)
        else
          match_result = result[0]
        end
        @logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#{@name}' and generated
'#{match_result.inspect}'") unless match_result.nil?
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
      end
    end

    return match_result
  end
end

class Parser

  attr_accessor :pos
  attr_reader :rules, :string, :logger

  class ParseError < RuntimeError
  end

  def initialize(language_name, &block)
    @logger = Logger.new(STDOUT)
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name = language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @tokens = []
    @string = string.clone
```

```
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
      raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |tok|
        match = tok.pattern.match(string)
        # The regular expression of a token has matched the beginning of 'string'
        if match
          @logger.debug("Token #{match[0]} consumed")
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
          true
        else
          # this token pattern did not match, try the next
          false
        end # if
      end # raise
    end # until
  end

  def parse(string)
    # First, split the string according to the "token" instructions given.
    # Afterwards @tokens contains all tokens that are to be parsed.
    tokenize(string)

    # These variables are used to match if the total number of tokens
    # are consumed by the parser
    @pos = 0
    @max_pos = 0
    @expected = []
    # Parse (and evaluate) the tokens received
    result = @start.parse
    # If there are unparsed extra tokens, signal error
    if @pos != @tokens.size
      raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found
'#{@tokens[@max_pos]}'"
    end
    return result
  end

  def next_token
    @pos += 1
    return @tokens[@pos - 1]
  end

  # Return the next token in the queue
  def expect(tok)
    t = next_token
    if @pos - 1 > @max_pos
      @max_pos = @pos - 1
      @expected = []
    end
    return t if tok === t
    @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
    return nil
  end

  def to_s
    "Parser for #{@language_name}"
  end

  private

  LexToken = Struct.new(:pattern, :block)

  def token(pattern, &block)
    @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
  end
```

```
  def start(name, &block)
    rule(name, &block)
    @start = @rules[name]
  end

  def rule(name,&block)
    @current_rule = Rule.new(name, self)
    @rules[name] = @current_rule
    instance_eval &block
    @current_rule = nil
  end

  def match(*pattern, &block)
    @current_rule.send(:match, *pattern, &block)
  end

end
```