Sharp-ruby

“En blandning av det bästa från C# och Ruby”

Linköpings Universitet

Innovativ Programmering

TDP019: datorspråk

Zackarias Ouacha - zacou776@student.liu.se

Michael Lake - micla389@student.liu.se

Utskriftsdatum: 2021-05-04

Sammanfattning

Språket Sharp-Ruby har genomförts på Linköpings universitet på programmet “Innovativ programmering” i kursen “TDP019 Projekt: Datorspråk” under perioden mars till juni år 2021. Vi fick i uppgift att skapa ett eget programspråk. Vårt språk bygger på idén att ta det bästa av två språk vi jobbat mycket med och har olika erfarenheter av, Ruby och C#. Rubys simplicitet och mjukhet blandat med precisionen av C#. Målgruppen för språket är någon som vill ha precisionen av C# men kanske tycker att C# inte är tillräckligt förlåtande. Vi har förenklat syntaxen men ändå bibehållit precisionen för att förenkla programmerandet men samtidigt behålla kvalitén. Sharp-Ruby är implementerat med hjälp av ett verktyg som heter RDparser som är skrivit i Ruby.

Innehållsförteckning

1. Inledning

Det här projektet har genomförts på Linköpings universitet, på programmet “Innovativ programmering” i kursen “TDP019 Projekt: Datorspråk” under perioden mars till juni år 2021. Dokumentationen består av fyra delar; inledning, användarhandledning, systemdokumentation och reflektion.

1.1 Syfte

Syftet med projektet var att vi skulle lära oss hur programspråk är uppbyggt för att på så sätt förstå vad som händer bakom kulisserna och bakgrunden till den programmering vi dagligen utför. Genom att skapa ett eget programspråk har vi lärt oss hur programkod parsas, körs och hur det är uppbyggt.

1.2 Bakgrund

Vårt språk bygger på idén att ta det bästa av två språk vi jobbat mycket med och har olika erfarenheter av, “Ruby” och “C#”. Rubys simplicitet och “mjukhet” blandat med precisionen av C# var något vi tyckte skulle vara väldigt intressant att blanda. Vi tänkte att det skulle resultera i ett språk som hamnar någonstans mittemellan när det gäller svårighetsgrad och precision.

1.3 Målgrupp

Målgruppen för språket är någon som vill ha precisionen av C# men kanske tycker att C# inte är tillräckligt förlåtande. Vi har förenklat syntaxen men ändå bibehållit precisionen för att förenkla programmerandet men samtidigt behålla kvalitén.

2. Användarhandledning

Installera Ruby, navigera sedan till mappen Sharp-Ruby. Längst ned i filen skriver du in vilken fil du vill köra inom parenteserna i kodraden, till exempel:

“*a.parse_test("FilNamn.rb")*”.

Navigera sedan till filen “SRParser.rb” i terminalen och skriv ruby SRParser.rb. Den filen som du skrivit in att du vill köra i koden i SRParser.rb kommer nu att köras.

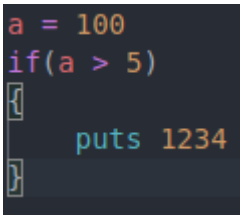
2.1 Konstruktioner

Konstruktioner innehar villkorssatser, repetitionssatser, datatyper, operator och funktioner.

2.1.1 Villkorssats

If-satser är i vårt fall den enda villkorssatsen i språket. If-satser används för att bestämma om ett villkor är uppfyllt, om villkoret är uppfyllt så ska en operation utföras. Detta är viktigt att ha i ett språk t.ex. då variabler kan ändras, och när en variabel har ändrats för att uppfylla ett visst villkor, så kanske man vill utföra en specifik operation vid just det tillfället.

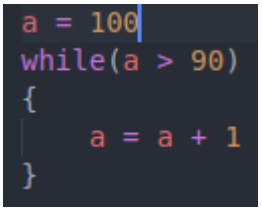
Sharp-rubys If-satser ser ut så här:



2.1.2 repetitionssats

While-loops är i vårt fall den enda repetitionssatsen i språket. While-loops används för att bestämma om ett villkor är uppfyllt och kör ett visst antal operationer medans villkoret är sant.

Sharp-rubys While-loops ser ut så här:



2.1.3 Datatyper

I vårt språk har vi datatyperna: float, int och bool

2.1.4 Operatorer

Sharp-ruby innehåller tre olika typer av operatorer:

1. Aritmetiska operatorer: De operatorer som används i aritmetisk matematik, alltså +, -, *, /, % och ( )

```
rule :primaryOperator do
    match("/") {|a | a}
    match("%") {|a | a}
    match("*") {|a | a}
end
```

```
rule :secondaryOperator do
    match("+") {|a | a}
    match("-") {|a | a}
end
```

2. Logiska operatorer: De operatorer som används för logiska uttryck, alltså and, or, == och !.
3. Jämförelseoperatorer: De operatorer för att jämföra något med något annat, alltså: >, <, >=, <=, and, or, == och !.

```
rule :compOperator do
    match(">") {|a| a}
    match("<") {|a| a}
    match(">=") {|a| a}
    match("<=") {|a| a}
    match("and") {|a| a}
    match("or") {|a| a}
    match("==") {|a| a}
    match("!") {|a| a}
end
```

```
rule :termOperator do
    match("and") {|a| a}
    match("or") {|a| a}
    match("==") {|a| a}
    match("!") {|a| a}
end
```

compOperator används för alla sorters datatyper förutom bools när det gäller logiska utvärderingar. *termOperator* används enbart för bools.

Parenteser ingår också i programspråket.

2.1.5 Tilldelning och variabler

I SharpRuby kan en variabel skrivas som en sträng som består av bokstäver och siffror, till exempel “A” eller “var1”. Tilldelning sker genom att skriva ett variabelnamn, följt av ett likamedtecken för att sedan följas av ett värde, en uträkning eller en annan variabel, exempelvis:

a = 100
var1 = 10 * 25
var2 = a + var1

2.1.6 Print

För att printa värdet av en variabel så kan “print” funktionen användas. Den används på så sätt att man skriver ordet “puts” följt av den variabel som man vill få ut värdet på. Ungefär såhär:

```
a = 100 + 11
puts a
```

Vilket kommer skriva ut 111.

2.1.7 Funktioner

Funktioner i SharpRuby är uppdelat och skrivs i tre delar: funktionsnamnet, parameterlista samt blocket med vad som ska utföras. Funktionsnamn skrivs med stor bokstav, den parametern eller de parametrar som ska skickas in skrivs inom parenteser, och kodblocket skrivs inom måsvingar. Ett exempel:

```
SumFunction(a, b, c)
{
        result = a + b + c
}
```

2.1.8 Scope

Scope eller räckviddshantering är ett sätt att hantera i vilket lager av ett program en variabel eller annan data lagras i. Detta är för att man inuti en funktion ska kunna till exempel skapa variabler utan att de påverkar de variabler som ligger i det globala scopet, ett lager “bakom” funktionen.

3. Systemdokumentation

3.1 Översikt

Språket SharpRuby bygger på användning av RDParse.rb för att göra den lexikaliska analys samt parsning av programmen som skrivs i SharpRuby-språket. Resultatet blir att ett nodsystem i form av ett syntaxträd skapas. Noderna är olika typer av klassobjekt som skapas och läggs in i rätt ordning i trädet.

3.2 Kodstandard

Eftersom att språket är baserat på att ta det bästa från C# och Ruby, blir kodstandarden för SharpRuby en kombination av de båda språken.

3.3 Grammatik

Grammatiken består av BNF-regler:

BNF

<program> ::= <runs>

<runs> ::= <runs><run> | <run>

<run> ::= <function> | <callFunction> | <print> | <whileLoop> | <ifOperation> | <assign> | <termExpressions> | <comparisonExpressions> | <expressionSec>

<function> ::= func<functionName>(<types>){<codeBlocks>}

<callFunction> ::= <functionName>(<types>)

<types> ::= <types><type> | <type>

<ifOperation> ::= if(<condition>){<statement>}

<whileLoop> ::= (condition){<statement>}

<condition> ::= <comparisonExpressions> | <term>

<assign> ::= <variableName> = <expressionSec> | <variableName> = <term>

<codeBlocks> ::= <codeBlocks><codeBlock> | <codeBlock>

<codeBlock> ::= <callFunction> | <print> | <whileLoop> | <ifOperation> | <assign> | <termExpressions> | <comparisonExpressions> | <expressionSec>

<print> ::= puts <variableName> | puts <type>

<comparisonExpressions> ::= <comparisonExpressions> <compOperator> <comparisonExpressions> | <expressionSec>

<termExpressions> ::= <term> <termOperator> <term> | <variableName> <termOperator> <term> | <term> <termOperator> <term> | <variableName> <termOperator> <variableName>

<expressionSec> ::= <expressionPrim> | <expressionSec> <secondaryOperator> <expressionPrim>

<expressionPrim> ::= <expressionPrim> <primaryOperator> <parentheses> | <expressionPrim> <primaryOperator> <type> | <type> | <parentheses>

<parentheses> ::= (<expressionSec>)

<type> ::= <int> | <float> | <variableName>

<term> ::= <bool>

<bool> ::= true | false

<compOperator> ::= < | > | == | <= | >= | != | and | or | !

<termOperator> ::= == | != | and | or | !

<secondaryOperator> ::= - | +

<primaryOperator> ::= * | / | %

<functionName> ::= <string>

<variableName> ::= <string>

<string> ::= /[a-zA-Z0-9]+/

<float> ::= <Float> | -<float>

<int> ::= <Integer> | -<int>

<Integer> ::= /[0-9]*/

4. Reflektion

Vi kom på idén till vårt språk genom att ha jobbat med både C# och Ruby, och tyckte att det fanns för- och nackdelar med båda. Vi kände därför att ett mellanting mellan språken hade varit väldigt optimalt då det hade bidragit till ett språk med den simpliciteten som Ruby har, men med en sida av direkthet och avancerad funktionalitet C# Vi hann dock inte tillämpa så mycket av sakerna som vi föredrar från C# i språket då de sakerna är mer avancerad funktionalitet, vilket vi tyvärr inte hann med.

Vårt mål var att hinna till den delen av språket där man hade sett kontrasten mellan de två inspirationsspråken och kunna se fördelarna av vår idé, och till en början låg vi väl inom den tidsramen vi hade tänkt ut och följde schemat. Vi stötte dock på problem som gjorde att vi hamnade efter, vilket gjorde att slutresultatet inte är där vi hade velat från början. Vi fastnade en del när vi skulle dela upp parsningsregler och funktionalitet i två filer, men det stora tidsförhindret kom när språket skulle kunna läsa in flera rader kod. Detta var något vi fann självklart och att det skulle ske automatiskt från början, dock insåg vi snabbt att ett rejält hinder stod framför oss. Det tog oss nästan två veckor att lösa problemet och efter det var det inte lång tid kvar till deadline. Även nu innan deadline är vi inte riktigt helt färdiga med funktionaliteten som krävs för kursen vilket är väldigt tråkigt. Vi hade velat implementera den väldigt specifika klassimplementationen med properties och vilket scope klassen tillhör som C# har, men det kommer väldigt sent in i ett språk; ett stadie vi aldrig lyckades nå. Vi tror att detta kunde ha motverkats med hjälp av tidigare kontakt med handledare, mer efterforskning och att ta större vara på den hjälp som finns, vare sig det är att kolla på tidigare språkdokumentation som finns, forum på internet eller fråga efter hjälp.

Trots att vi inte riktigt är klara så kan vi se på detta negativt men också positivt. Språket är inte där vi hade velat i slutskedet, och vi hade iallafall velat ligga på godkändnivå till deadline för inlämning. Vi ser dock positivt på att vi lyckades fixa ett problem som tog så lång tid att dels felsöka, men också lång tid att lösa. Vi gav aldrig upp och gjorde allt vi kunde för att lösa det vilket vi ser som en prestation i sig. Vi hoppas kunna lösa det sista under kompletteringsstadiet av språket för att lyckas nå ett betyg på projektdelen i kursen.

5. Bilagor

5.1 SRParser.rb

```
require './rdparse.rb'
require './nodes.rb'

class SRParser
   def initialize
      @parser = Parser.new("SharpRuby") do

         #To create a VariableList class to be able to store all variables
         variables = VariableList.new()

         #To create an Operations class to be able to store all operations
         operations = Operations.new()

         #List to store all parsed statements in
         @@statementList = []

         #List to store codeblocks in
         @@blockList = []

         #List to store function parameters in
         @@parameterList = []

         ######## Parsing Tokens ########

         token(/^?\s+/) #Whitespace characters (space, return)

         token(/^?\d+\.\d+/) {|x| x.to_f} #Floats

         token(/^?\d+/) {|int| int.to_i} #Integers

         token(/^?[a-zA-Z0-9]+/) {|m| m.to_s} #Characters, numbers and words that
         define a variable name

         token(/[><=!+]+/) {|m| m}

         token(/[==]+/) {|m| m } #The logical operator of "=="

         token(/^?./) {|m| m } #Everything else
```

######## PARSING RULES ########

```
start :program do
  match(:runs)
end

rule :runs do
  match(:runs, :run) {|_, a| @@statementList << a}
  match(:run) {|a| @@statementList << a}
end

rule :run do
  match(:function)
  match(:print)
  match(:whileLoop)
  match(:ifOperation)
  match(:assign)
  match(:termExpressions)
  match(:comparisonExpressions)
  match(:expressionSec)
end

rule :function do
   match("func", :functionName, "(",:types, ")", "{", :codeBlocks, "}") {|_, name, _,
            parameter, _, _, codeblock, _| FunctionNode.new(name,
            @@parameterList, codeblock)}
end

rule :callFunction do
  match(:functionName, "(" ,:types,")") {|a,_,b,_| }
end

rule :types do
  match(:types, :type) {|_,a| @@parameterList << a}
  match(:type) {|a| @@parameterList << a}
end

rule :whileLoop do
   match("while", "(", :term , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_|
            operations.add("while", a,b,nil,variables)}
  match("while", "(", :comparisonExpressions , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_|
            operations.add("while", a,b,nil,variables)}
```

```
    match("if", "(", :termExpressions , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_|
            operations.add("if",a,@@blockList, nil,variables)}
  end

  rule :ifOperation do
    match("if", "(", :term , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_| operations.add("if",a,
            @@blockList, nil,variables)}
    match("if", "(", :comparisonExpressions , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_|
            operations.add("if",a,@@blockList, nil,variables)}
    match("if", "(", :termExpressions , ")", "{", :codeBlocks, "}") {|_,_,a,_,_,b,_|
            operations.add("if",a,@@blockList, nil,variables)}
  end

  rule :codeBlocks do
    match(:codeBlocks, :codeBlock) {|ops,op| BlockList.new(ops, op)}
    match(:codeBlock) {|op| Block.new(op)}
  end

  rule :codeBlock do
    match(:whileLoop)
    match(:ifOperation)
    match(:assign)
    match(:termExpressions)
    match(:comparisonExpressions)
    match(:expressionSec)
    match(:print)
  end

  rule :print do
    match("puts", :variableName) {|_, a| Print.new(variables.get(a))}
    match("puts", :type) {|_, a| Print.new(a)}
  end

  rule :assign do
    match(:variableName, "=", :expressionSec) {|a,_,b| variables.add(a,b)}
    match(:variableName, "=", :term) {|a,_,b| variables.add(a,b)}
  end

  rule :comparisonExpressions do
    match(:comparisonExpressions, :compOperator, :comparisonExpressions) {|a,b,c|
            operations.add("comp",a,b,c,variables)}
    match(:expressionSec)
  end
```

```
rule :termExpressions do
    match(:term, :termOperator, :term) {|a,b,c| operations.add("comp",a,b,c,variables)}
    match(:variableName, :termOperator, :term) {|a,b,c|
            operations.add("comp",a,b,c,variables)}
   match(:term, :termOperator, :term) {|a,b,c| operations.add("comp",a,b,c,variables)}
    match(:variableName, :termOperator, :variableName) {|a,b,c|
            operations.add("comp",a,b,c,variables)}
end

rule :expressionSec do
   match(:expressionPrim)
    match(:expressionSec, :secondaryOperator, :expressionPrim) {|a, b, c|
            operations.add("op",a,b,c,variables)}
end

rule :expressionPrim do
    match(:expressionPrim, :primaryOperator, :parentheses) {|a, b, c|
            operations.add("op",a,b,c,variables)}
    match(:expressionPrim, :primaryOperator, :type) {|a, b, c|
            operations.add("op",a,b,c,variables)}
   match(:type)
   match(:parentheses)
end

rule :parentheses do
   match("(", :expressionSec, ")") {|_,b,_| operations.add("par",nil, b, nil,variables)}
end

rule :primaryOperator do
   match("/") {|a | a}
   match("%") {|a | a}
   match("*") {|a | a}
end

rule :secondaryOperator do
   match("+") {|a | a}
   match("-") {|a | a}
end

rule :compOperator do
   match(">") {|a| a}
   match("<") {|a| a}
```

```
    match(">=") {|a| a}
    match("<=") {|a| a}
    match("and") {|a| a}
    match("or") {|a| a}
    match("==") {|a| a}
    match("!") {|a| a}
  end

  rule :termOperator do
    match("and") {|a| a}
    match("or") {|a| a}
    match("==") {|a| a}
    match("!") {|a| a}
    match("!=") {|a| a}
  end

  rule :type do
    match(:float)
    match(:int)
    match(:variableName) {|a| a}
  end

  rule :variableName do
    match(/[a-zA-Z0-9]+/) {|a| VarName.new(a).get_varName}
  end

  rule :functionName do
    match(/[a-zA-Z0-9]+/) {|a| a}
  end

  rule :string do
     match(String) {|a|a}
  end

  rule :float do
    match(Float) {|a| Float.new(a)}
    match("-", :float) {|_,a | Float.new(-a)}
  end

  rule :int do
    match(Integer) {|a| Int.new(a)}
    match("-", :int) {|_,a | Int.new(-a)}
  end
```

```
        rule :term do
            match(:bool)
        end

        rule :bool do
            match('true') {|a| Bool.new(a)}
            match('false') {|a| Bool.new(a)}
        end
    end

    def done(file_data)
        ["quit","exit","bye",""].include?(file_data.chomp)
    end

    #Function for testing
    def parse_test(fileName)
        print "[SRParser] "
        file = File.open(fileName)
        file_data = file.read

        parseData = @parser.parse(file_data)
        newData = Statements.new(@@statementList)

        file.close
    end
end
end

a = SRParser.new
a.parse_test("testFILE.rb")
```

5.2 Nodes.rb

```
####### TYPES #######
class Int

   def initialize(value)
     @value = value
   end

   def eval
     return @value
   end
end

class Float

   def initialize(value)
      @value = value
   end

   def eval
      return @value
   end
end

######## TERM ########

class Bool

   def initialize(value)
      @value = value
   end

   def eval
      return @value
   end
end

######## PRINT FUNCTION ########

class Print

   def initialize(value)
```

```
    @value = value
  end

  def eval()
    puts @value.eval
    @value.eval
  end
end



class Statements

  def initialize(list)
    @list = list
    @evalList = []
    eval
  end

  def eval
    for stuff in @list do
      if(stuff.class == Array)
        for nodes in stuff
          if(nodes.class == Variable)

            @evalList << nodes.get_value.eval
          end
        end
      else
        @evalList << stuff.eval
      end
    end
    puts @evalList
    return @evalList
  end
end


class Operations

  def initialize()

  end
```

```
    def add(type, a,b,c, variables)
        #puts "class"
        #puts a.class

        if(a.class == String)
            #puts "SEC"
            #puts variables.get_var_value(a).eval
            a = variables.get_var_value(a)
        elsif(b.class == String)
            b = variables.get_var_value(b)
        end
        if(type == "comp")
            comp = Comparison.new(a,b,c)
        elsif(type == "op")
            op = Operator.new(a,b,c)
        elsif(type == "par")
            par = Parentheses.new(b)
        elsif(type == "if")
            ifOp = IfOperation.new(a,b)
        elsif(type == "while")
            whileOP = While.new(a,b)
        end
    end
end

######## CODE BLOCK HANDLING ########

class Block

    attr_accessor :operation

    def initialize(operation)
        @operation = operation
    end

    def eval()
        @operation.eval
    end
end

class BlockList
```

```
    attr_accessor :operations

    def initialize(operation, operations)
        @operations = [operation]
        @operations << operations
    end

    def eval()
        #call eval on every operation in collection of operations
        @operations.each {|op| op.eval}
    end
end

######## ITERATION ########

class While

    def initialize(expr, block)
        @expr = expr
        @block = block

    end

    def eval
        while @expr.eval
            @block.eval

        end
    end
end

class IfOperation

    def initialize(expr, blockList)
        @expr = expr
        @blockList = blockList
        @evalList = []
     end

     def eval
        if(instance_eval("@expr") or @expr.eval)
            for stuff in @blockList do
                @evalList << stuff.eval
```

```
            end
        else
            return false
        end

        return @evalList
    end
end

######## FUNCTION HANDLING ########

@@functions = {}


class FunctionNode

    def initialize(name, para, block)
        @name = name
        @paraList = para
        @block = block
    end

    def eval
        @@functions[name] = [@paraList, @block]
    end
end

class CallFunction

    def initialize(name, para)
        @name = name
        @para = para
    end
end

######## VARIABLE HANDLING ########

class VariableList

    def initialize()
        @@varList = []
    end
```

```
  def get_var_value(varName)
    for var in @@varList do
      if(var.get_name == varName)
        return var.get_value
      end
    end
  end

  def get_var_name(value)
    for var in @@varList do
      if(var.get_name == varName)
        return var.get_name
      end
    end
  end

  def delete(oldValue, name)
    for node in @@varList do
      if(node.get_value.eval == oldValue && node.get_name == name)
        @@varList.delete_at(@@varList.index(node))
      end
    end
  end

  def add(varName, value)
    if(@@varList == [])
      variable = Variable.new(varName, value)
      @@varList << variable
    else
      for var in @@varList do
        if (var.get_name == varName)
          oldValue = var.get_value.eval
          varName.set_value(value)
          delete(oldValue, varName)
        else
          puts "BLA"
          variable = Variable.new(varName, value)
          @@varList << variable
          return
        end
      end
    end
  end
```

```
    def get_list
        return @@varList
    end
end

class VarName

    def initialize(varName)
        @varName = varName
    end

    def get_varName
        return @varName
    end

end

class Variable

    def initialize(varName, value)
      @varName = varName
      @value = value
    end

    def get_name
        return @varName
    end

    def get_value
        return @value
    end

    def set_value(value)
        @value = value
    end

    def eval
        return instance_eval("#{@varName} = #{@value}")
    end
end

######## MATHEMATICS AND LOGIC ########
```

```
class Operator

    def initialize(var1, operator ,var2)
        @var1 = var1.eval
        @var2 = var2.eval
        @operator = operator
    end

    def eval()
        if(@var2.class == String or @var1.class == String)
            return instance_eval("@var1 #{@operator} @var2")
        else
            return instance_eval("#{@var1} #{@operator} #{@var2}")
        end
    end
end

class Comparison

    def initialize(term1, compOperator, term2)
        @term1 = term1.eval
        @term2 = term2.eval

        @compOperator = compOperator
    end

    def eval()
        return instance_eval("@term1 #{@compOperator} @term2")
    end
end

class Parentheses

    def initialize(expr)
        @expr = expr.to_s
    end

    def eval()
        return instance_eval("#{@expr}")
    end
end
```