VANILLA

Ett programmeringsspråk byggt i kursen
TDP019-Projekt: Datorspråk

Love Jansson: lovja643@student.liu.se

Innovativ programmmering
Linköpings Universitet
3 juni 2020

Contents

1 Inledning

Detta dokument är en del av examinationen i projektkursen TDP019 där vi skapat ett datorspråk. Målet med Vanilla var att skapa ett generellt programmeringsspråk med syntax som vi uppskattat från de språk vi har erfarenhet av sedan tidigare; C++, Python och Ruby.

Ett program skrivet i Vanilla består av en eller flera funktioner varav huvudfunktionen chief anropas vid körning av programmet. Inom funktionerna finns möjlighet att implementera sedvanliga konstruktioner såsom deklarering och tilldelning av variabler, if-satser, och repetitionssatser. Funktionerna kan också anropas rekursivt. Utöver vanliga funktioner finns också lambda-funktioner. De datatyper som finns i språket är Int, Float, String, Bool, List och Map. Språket är typat vilket innebär att korrekt datatyp ska specificeras vid deklarering av variabler samt för returvärde vid funktionsdefinitioner. Dokumentet kan vidare delas in i tre delar:

1. **Användarhandledning**. En komma igång-guide där konstruktioner och syntax i språket beskrivs.
2. **Systemdokumentation**. Här förklaras hur systemet är uppbyggt, dvs., hur koden är strukturerad, samt grammatiken för språket.
3. **Reflektioner**. I det sista avsnittet återges tankar och lärdomar som upkommit under projektets gång.

2 Användarhandledning

Nedan följer en komma igång-guide som beskriver hur språket används. För att kunna skriva program i Vanilla krävs att programmeringsspråket Ruby installeras. Mer information finns på följande länk:
https://www.ruby-lang.org/en/documentation/installation/.
Gå sedan in på https://gitlab.liu.se/lovja643/vanilla och ladda ned mappen vanilla.zip.

Nu har du allting som krävs för att skriva ett Vanilla-program. Språket används genom att du skapar en fil med filändelsen ".v" och kör filen med hjälp av kommandot "ruby vanilla.rb filnamn". Om filen ligger i samma mapp som vanilla.rb (som ligger i vanilla.zip) behöver du inte ange sökväg. Om filen ligger i annan mapp ska sökväg anges.

2.1 Ditt första program

I varje Vanilla-program måste det finnas en funktion som heter chief. Detta är huvudfunktionen utifrån vilken hela programmet evalueras. Ett felmeddelande kastas om flera chief-funktioner definierats eller om den utelämnas. För att ge ett första exempel på hur språket används skapar vi ett program som skriver ut "Hello there" i terminalen. Följ nedanstående steg:

1. **Skapa fil**. Skapa en fil med valfritt namn och filändelsen ".v". Spara filen i mappen Vanilla som du hämtade ned från gitlab.
2. **Redigera fil**. Öppna filen och skriv in den kod som finns i programexempel 1 nedan.
3. **Kör filen**. Öppna en terminal och navigera till mappen vanilla. Kör filen med kommando "ruby vanilla.rb filnamn". Nu ska meddelandet "Hello there" skrivas ut i terminalen.

```
1 nothing chief():
2
3     print("Hello there")
4
5 end
```

Programexempel 1: Hello there.

Innan en mer detaljerad genomgång av språket ges nämns först att kommentarer skrivs genom att påbörja meningen med '#'. Det går tyvärr inte att skriva flerradskommentarer i nuläget.

2.2 Datatyper

De datatyper som finns i Vanilla är Int, Float, String, Bool, List och Map. I detta avsnitt beskrivs de metoder datatyperna har samt hur de kan lagras i variabler genom deklarering och tilldelning.

2.3 Int

Programexempel 2 visar hur Int lagras i variabler. När en Int-variabel deklareras utan värde lagras värdet 0.

```
1  nothing chief():
2       #Deklarering
3       int x = 1
4       int y
5
6       #Tilldelning
7       y = 2
8
9  end
```

Programexempel 2: deklarering och tilldelning av Int.

De metoder som tillhör Int är type(), absolute(), string() och float(). Ingen av dessa metoder ändrar på objektet. Programexempel 3 nedan visar hur metoderna används.

```
1  nothing chief():
2       int x = -1
3
4       x.type()        # => "Int"
5       x.absolute()    # => 1
6       x.string()      # => "-1"
7       x.float()       # => "-1.0"
8
9  end
```

Programexempel 3: metoder till Int.

2.4 Float

Programexempel 4 visar hur Float lagras i variabler. När en Float-variabel deklareras utan värde lagras värdet 0.0.

```
1  nothing chief():
2       #Deklarering
3       float x = 1.1
4       float y
5
6       #Tilldelning
7       y = 2.2
8
9  end
```

Programexempel 4: deklarering och tilldelning av Float.

Float har metoderna type(), absolute(), och string() precis som Int. Utöver dessa tillkommer int() och round(). Ingen av dessa metoder ändrar på objektet. Programexempel 5 visar hur de sistnämnda metoderna används.

```
1  nothing chief():
2       float x = 1.16
3
4       x.round()  # => 1
5       x.round(1) # => 1.2
6       x.int()    # => 1
7
8  end
```

Programexempel 5: metoder till Float.

2.5 String

Programexempel 6 visar hur String lagras i variabler. När en String-variabel deklareras utan värde lagras en tom sträng(""). Strängar omges antingen av " eller '.

```
1  nothing chief():
2       #Deklarering
3       string x = "Love"
4       string y
5
6       #Tilldelning
7       y = 'Emma'
8
9  end
```

Programexempel 6: deklarering och tilldelning av String.

String har 12 metoder. Utav dessa är det enbart metoden ”[]=” som ändrar på objektet. Programexempel 7 illustrerar metodernas användning.

```
nothing chief():
    string x = "Love"
    string y = "4"

    #Anropar metoder via x
    x.type()            # => "String"
    x[0]                # => "L"
    x[0] = "T"          # => "Tove"
    x.length()          # => 4
    x.split()           # => ["T", "o", "v", "e"]
    x.upper()           # => "TOVE"
    x.lower()           # => "tove"
    x.has_chr("e")      # => true
    x.sub("e", "3")     # => "Tov3"

    #Anropar metoder via y
    y.int()             # => 4
    y.float()           # => 4.0
    y.is_alpha()        # => false

end
```

Programexempel 7: metoder till String.

2.6 Bool

Programexempel 8 visar hur Bool lagras i variabler. När en Bool-variabel deklareras utan värde lagras värdet false.

```
1  nothing chief():
2       #Deklarering
3       bool x = true
4       bool y
5
6       #Tilldelning
7       y = false
8
9  end
```

Programexempel 8: deklarering och tilldelning av Bool.

Bool har enbart en metod; type(), vilken returnerar strängen "Bool".

2.7 List

Programexempel 9 visar hur List lagras i variabler. När en List-variabel deklareras utan värde lagras en tom lista ([]).

```
1  nothing chief():
2       #Deklarering
3       list<int> x = [1, 2, 3]
4       list<list<int>> y
5
6       #Tilldelning
7       y = [[1, 2], [3, 4]]
8
9  end
```

Programexempel 9: deklarering och tilldelning av List.

List har 9 metoder och utav dessa är det 4 som ändrar på objektet i fråga: []=, add_to_back, add_to_front och pop_at. Programexempel 10 illustrerar metodernas användning.

```
nothing chief():
    list<int> x = [1, 2, 3]

    #Anropar metoder via x
    x.type()              # => "List"
    x[0]                  # => 1
    x[0] = 10             # => [10, 2, 3]
    x.length()            # => 3
    x.sorted()            # => [2, 3, 10]
    x.add_to_front(0)     # => [0, 10, 2, 3]
    x.add_to_back(4)      # => [0, 10, 2, 3, 4]
    x.pop_at(0)           # => 0, (x == [10, 2, 3, 4])
    x.get_index(1)        # => 2

end
```

Programexempel 10: metoder till List.

När metoden sorted används via listor som innehåller listor sorteras listan baserat på det första elementet i den inre listan. Det är i nuläget inte möjligt för användaren att specificera detta själv.

2.8 Map

Programexempel 11 visar hur Map lagras i variabler. När en Map-variabel deklareras utan värde lagras en tom map ({}).

```
nothing chief():
     #Deklarering
     map<int, string> x = {1: "one", 2: "two"}
     map<list<int>, string> y

     #Tilldelning
     y = {[1, 2, 3]: "onetwothree"}

end
```

Programexempel 11: deklarering och tilldelning av Map.

Map har 6 metoder och utav dessa är det en ([]=) som ändrar på objektet i fråga. Programexempel 12 visar hur metoderna används.

```
1  nothing chief():
2       map<int, string> x = {1: "one", 2: "two"}
3
4      #Anropar metoder via x
5      x.type()              # => "Map"
6      x[1]                  # => "one"
7      x[3] = "three"        # => {1: "one", 2: "two", 3: "
       three"}
8      x.length()            # => 3
9      x.get_keys()          # => [1, 2, 3]
10     x.get_values()        # => ["one", "two", "three"]
11
12 end
```

Programexempel 12: metoder till Map.

2.9 Operatorer

I Vanilla finns logiska, aritmetiska, och jämförande operatorer. En sammanställning av dessa finns i bilden nedan. De är skrivna i prioritetsordning.

```
1      ^                          # => Aritmetisk
2      *, /, %                    # => Aritmetiska
3      +, -                       # => Aritmetiska
4      >, <, >=, <=, !=, ==       # => Jämförande
5      !, not, and, or            # => Logiska
```

Operatorer

Det finns även short hand operatorer i Vanilla. Se bilden nedan.

```
       +=          x += 2       # ersätter: x = x + 2
       -=          x -= 2       # ersätter: x = x - 2
       *=          x *= 2       # ersätter: x = x * 2
       /=          x /= 2       # ersätter: x = x / 2
       ^=          x ^= 2       # ersätter: x = x ^ 2
       %/          x %= 2       # ersätter: x = x % 2
```

Short hand operatorer

De logiska, aritmetiska och jämförande operatorerna kan inte appliceras på alla datatyper i språket varav en kort genomgång av dem blir nödvändig.

2.9.1 Logiska uttryck

De logiska operatorerna kan appliceras på alla datatyper. Operatorn för konjunktion är 'and'. När en konjunktion evalueras till falskt returneras Bool-värdet

false och om resultatet är sant returneras det sista sanna värdet. Detta innebär att returvärdet i det senare fallet inte nödvändigtvis är av datatypen Bool utan kan även vara någon av de andra datatyperna i språket. För att förklara ges ett exempel:
Anta att listan list<int>l = [1, 2, 3] finns i programmet. Uttrycket ”l and true” returnerar i detta fall true eftersom true står skrivet sist. Om uttrycket istället var ”true and l” returneras listan [1, 2, 3].

Operatorn för disjunktion är ’or’ och returnerar likt konjunktion Bool-värdet false när uttrycket är falskt. Till skillnad från konjunktion returneras det första sanna värdet i ett uttryck som är sant.

Operatorerna ! och not används för negation av ett logiskt uttryck. Resultatet är alltid i form av datatypen Bool.

2.9.2 Jämförande uttryck

Datatyperna String, Float och Int har samtliga jämförande operatorer. Vid storleksjämförelser (<, >, <=, >=) jämförs String lexikografiskt medan Float och Int jämförs enligt vanlig matematisk storleksordning. Operatorerna (==, !=) avgör om vänsterled och högerled har samma värde eller ej. Datatyperna List, Map och Bool har enbart de sistnämnda operatorerna.

2.9.3 Aritmetiska uttryck

De aritmetiska operatorerna kan tillämpas på datatyperna Int och Float enligt de matematiska räknereglerna. När det gäller datatypen String kan operatorn * och + förekomma. Det är möjligt att konkatenera strängar genom att använda ’+’, till exempel: ”love ” + ”Jansson” = ”love Jansson”. Det går även att multiplicera strängar med datatypen Int eller Float. Det numeriska värdet kan då placeras i vänsterled eller högerled. Det är alltså möjligt att skriva: 3 * ”hej” vilket evalueras till ”hejhejhej” samt ”hej” * 4.4 vars resultat blir ”hejhejhejhej”. De aritmetiska operatorerna kan inte appliceras på datatyperna Bool, List eller Map.

2.10 In- och utmatning

För in- och utmatning från/till terminalen används de inbyggda funktionerna input(), print() och printn() där printn även skriver ut en ny rad efter de angivna argumenten har skrivits ut i terminalen. Se programexempel 13.

```
1 nothing chief():
2       int num = input("Mata in ditt favoritnummer: ")
3
4       printn("Ditt favoritnummer är: ")
5
6       print(num)
7
8 end
9 # printn skriver även ut \n
```

Programexempel 13: In- och utmatning i terminal.

2.11 If-satser

If-satser fungerar i Vanilla i likhet med andra generella programmeringsspråk. Koden inom ett if-block evalueras om villkoret associerat med blocket uppfylls före villkoren för de andra blocken. Det går även att skriva nestade if-satser. Syntaxen ser ut som i programexempel 14 nedan.

```
1  nothing chief():
2      list<int> l = [1, 2, 3]
3
4      if l.length() == 2:
5          print("Längden är 2")
6      elseif l.length() == 1:
7          print("Längden är 1")
8      else:
9          print("Längden är 3")
10     endif
11
12 end
```

Programexempel 14: if-sats.

2.12 Repetitionssatser

I Vanilla finns tre olika repetitionssatser: for-loop, for-each samt while-loop. Dessa beskrivs i de delavsnitt som följer.

2.12.1 For-loop

En for-loop evaluerar koden inom det tillhörande blocket så många gånger som loopen anger. En iteratorvariabel av datatypen Int deklareras och tillges successivt ett högre värde för varje gång blocket har evaluerats. Steglängd specificeras genom en tilldelning av iteratorvariabeln. Programexempel 15 visar hur två enkla for-loopar kan se ut.

```
1  nothing chief():
2      int num1
3      int num2
4
5      for int i, i < 10, i += 1:
6          num1 += 1
7      endfor
8
9      for int i, i < 10, i += 2:
10         num2 += 1
11     endfor
12
13     # num1 == 10 och num2 == 5 på grund av olika steglängd (1 vs. 2) i looparna
14
15 end
```

Programexempel 15: for-loop.

Tänk på att ange ett korrekt uttryck samt en korrekt tilldelning av iteratorvariabeln. Annars kan du stöta på en oändlig loop.

2.12.2 For-each

En for-each itererar över en String, List eller Map. Den evaluerar koden inom det tillhörande blocket lika många gånger som antalet karaktärer i strängen, element i listan eller key-value-par i mappen. Steglängd är alltid 1 varav nyckelordet next (se rubrik 2.13) måste användas om användaren vill hoppa över ett varv.

När en for-each används finns möjligheten att använda datatypen auto vilket egentligen inte är en datatyp utan innebär att programmet tar reda på vilken datatyp iteratorvariabeln/iteratorvaraiblerna är. Nedan visas tre figurer för de olika for-each som finns. Programexempel 16 visar for-each för String, 17 för List och 18 för Map.

```
1  nothing chief():
2      string name = "Emma"
3
4      for string l in name: #'string l' kan ersättas med
       'auto l'
5          printn(l)
6      endfor
7
8  end
9  # Skriver ut följande i terminalen:
10 E
11 m
12 m
13 a
```

Programexempel 16: for-each String.

```
1  int chief():
2      list<int> l = [1, 2, 3]
3      int res
4
5      for int num in l: #'int num' kan ersättas med '
      auto num'
6          res += num
7      endfor
8
9  end
10 # returnerar 6
```

Programexempel 17: for-each List.

```
int chief():
    map<int, string> m = {1: "one", 2: "two", 3: "three"}

    for int n, string s in m: #'int n, string s' kan ersättas med 'auto n, auto s'
        printn(s + ": " + n.string())
    endfor

end
#Skriver ut följande i terminalen:
one: 1
two: 2
three: 3
```

Programexempel 18: for-each Map.

2.12.3 While-loop

En while-loop evaluerar koden inom det tillhörande blocket i omgångar tills det angivna uttrycket evalueras till false. Programexempel 19 visar hur syntaxen ser ut.

```
int chief():
    int num = 10

    while num > 0:
        num -= 1
    endwhile

    return num

end
# returnerar 0
```

Programexempel 19: while-loop.

2.13 break och next

Nyckelorden break och next finns tillgängliga för alla 3 versioner av repetition. break avbryter loopen medan next går vidare till nästa varv utan att evaluera resterande kod. Nedan i programexempel 20 och 21 visas hur break och next kan användas.

```
1  int chief():
2      list<list<int>> x = [[1, 2], [3, 4]]
3      int res
4
5      for list<int> l in x:
6          for int num in l do:
7              if num % 2 != 0:
8                  break
9              else:
10                 res += num
11             endif
12         endfor
13     endfor
14
15     return res
16
17 end
18 #returnerar 0 eftersom break i den innersta for-each
     loopen avbryter loopen innan elementen 2 och 4
```

Programexempel 20: break.

```
int chief():
    list<list<int>> x = [[1, 2], [3, 4]]
    int res

    for list<int> l in x:
        for int num in l do:
            if num % 2 != 0:
                next
            else:
                res += num
            endif
        endfor
    endfor

    return res

end
#returnerar 6 eftersom next inte avbryter den innersta
    loopen utan forsätter på nästa varv
```

Programexempel 21: next.

2.14 Funktioner

Inledningsvis nämndes att ett program i Vanilla består av en eller flera funktioner där exakt en huvudfunktion chief måste defineras. Funktionen chief har visats i programexempel i tidigare avsnitt. Det går även att definera fler funktioner på samma format som chief samt lambda-funktioner. Dessa två konstruktioner beskrivs i de två avsnitt som kommer härnäst.

2.14.1 Vanliga funktioner

När vanliga funktioner definieras är det viktigt att rätt datatyp anges för returvärde och parametrar. Om funktionen inte ska returnera någonting anges returtypen nothing. Alla funktioner som har ett returvärde måste innehålla ett return-statement, dvs., nyckelordet return följt av ett uttryck som anger returvärdet. Tvärtom får inte funktioner med returtypen nothing innehålla return-statements.

Syntax för hur en funktion defineras och anropas visas i programexempel 22 där fakulteten beräknas.

```
1  int chief():
2      int res = factorial(5)
3
4      return res
5
6  end
7
8  int factorial(int n):
9      if n == 1:
10         return n
11     else:
12         return n * factorial(n-1)
13     endif
14
15 end
16
17 #både factorial och chief returnerar 120
```

Programexempel 22: funktion factorial.

Programexempel 23 visar att parametrarna kan deklareras med default-värden. Observera att ordningen som parametrarna står i kan spela roll om en eller flera parametrar har default-värden. I exemplet nedan måste ett argument anges för parameter 'n' om parameter 's' ska kunna tillges ett värde. Programmet tolkar det andra funktionsanropet i exemplet som att 'n' ska tillges 'emma' varav ett fel kastas eftersom n är en Int-variabel. Det första Funktionsanropet fungerar eftersom 'n' tillges värdet 5 och 's' får sitt default-värde 'love'.

```
1  string chief():
2      string res1 = foo(5)
3
4      # kastar fel för inkorrekt datatyp
5      string res2 = foo("emma")
6
7      return res1
8
9  end
10
11 string foo(int n = 2, string s = "love"):
12    return n * s
13
14 end
```

Programexempel 23: default-värden för parametrar.

Vanilla har inte stöd för definition av vanliga funktioner i funktionskroppen och det är inte möjligt att ha parametrar som är vanliga funktioner. Det går dock att definera och returnera lambda-funktioner samt ha parametrar som är lambda-funktioner (se nästa avsnitt 2.14.2). Programexempel 24 och 25 visar exempel på detta.

```
1  int chief():
2      int res = foo(5, [](int y): y + 4 end)
3
4      return res
5
6  end
7
8  int foo(int n, lambda f):
9          return f(n)
10
11 end
12 #chief returnerar 9.
```

Programexempel 24: lambda-parameter i vanlig funktion

```
1  int chief():
2      lambda adder = create_adder()
3
4       return adder(2, 2)
5  end
6
7  lambda create_adder():
8      return [](int num1, int num2): num1 * num2 end
9  end
10
11 chief returnerar 4
```

Programexempel 25: vanlig funktion som returnerar lambda.

2.14.2 Lambda

Syntax för en lambda-funktion är: **[captures](parametrar): uttryck end**. Det är alltså en funktion som inte har något namn och enbart får innehålla ett uttryck. Funktionskroppen kan innehålla ett aritmetiskt-, logiskt eller -jämförande uttryck. Den kan även innehålla en lambda-definition eftersom det är ett uttryck och inte en sats. Funktionskroppen returnerar resultatet av det uttryck som står skrivet. Ingen datatyp för returvärde behöver anges för lambda-funktioner som vid vanliga funktionsdefinitioner men parametrarnas datatyper måste anges. Precis som vid vanliga funktionsanrop går det också bra att ange default-värden för parametrarna i lambda-funktioner.

Syntaxen för lambda funktioner visas i programexempel 26 nedan. Bilden visar hur en lambda funktion lagras i en variabel samt anropas ungefär som vid vanliga funktionsanrop, men istället för att ange namnet på funktionen är det variabelnamnet som anges. I nuläget går det bara att anropa lambda-funktioner genom att lagra dem i en variabel. Vid funktionsanrop har vanliga funktioner högre prioritet än lambda-funktioner vilket innebär att om en vanlig funktion och en lambda-funktion har samma namn är det den vanliga funktionen som anropas.

```
int chief():
    lambda x = [](int y): y + factorial(y) end

    return x(5)

end

int factorial(int n):
    if n == 1:
        return n
    else:
        return n * factorial(n-1)
    endif

end

#factorial returnerar 120, chief returnerar 125
```

Programexempel 26: lambda med ett aritmetiskt uttryck.

Programexempel 27 visar att en lambda-funktion kan ha lambda-parametrar. Observera att det inte är möjligt att ha parametrar som är vanliga funktioner.

```
int chief():
    lambda func = [](int n, lambda f): f(n) end
    int res = func(5, [](int y): y + 4 end)

    return res

end

#chief returnerar 9.
```

Programexempel 27: lambda-parameter.

I vanliga fall har inte funktionskroppen i en lambda tillgång till det scope där funktionen definerats eller det scope där funktionen anropas. Det är bara parametrarna som kan nås. Om användaren vill att programmet ska ha tillgång till andra variabler under ett anrop kan captures användas, vilket är en lista med variabelnamn som ska finnas tillgängliga i scope under ett anrop.

Programexempel 28 visar hur captures används genom att variabeln 'n' finns med i det scope som funktionskroppen har tillgång till. I samband med detta visas även att en lambda-funktion kan skapa och returnera en lambda-funktion. Observera att det inte går att definera vanliga funktioner i en lambda-funktion.

```
1  int chief():
2      lambda l = [](int n): [n](int num): num * n end
       end
3
4      lambda doubler = l(2)
5
6      return doubler(2)
7
8  end
9
10 # chief returnerar 4
```

Programexempel 28: lambda-funktion som returnerar lambda + captures.

2.15 Scope

Beroende på vilken del av koden programmet evaluerar är räckvidden till variabler olika. Varje vanlig funktion bildar ett eget scope och när koden inom funktionen evalueras har programmet tillgång till de variabler som deklareras där. Samtidigt kan inte de variabler som deklarerats i andra funktioner nås.

Vidare bildar varje if-sats och repetitionssats ett scope inom funktionens scope. När programmet evaluerar dessa scope har det tillgång till både variabler inom satsens scope samt i funktionens scope. Varje ny nivå av scope har tillgång till alla tidigare nivåer inom samma funktion.

När variabler ska nås eller tillges nya värden och det finns variabler med samma namn på olika nivåer av scope är det den mest lokala variabeln vars värde ändras/nås, dvs., variabeln i det senast tillagda scopet. Programexempel 29 nedan visar exempel på hur scope fungerar.

Angående lambda-funktioner förklarades i föregående avsnitt att de har tillgång till parametrarna och eventuella captures.

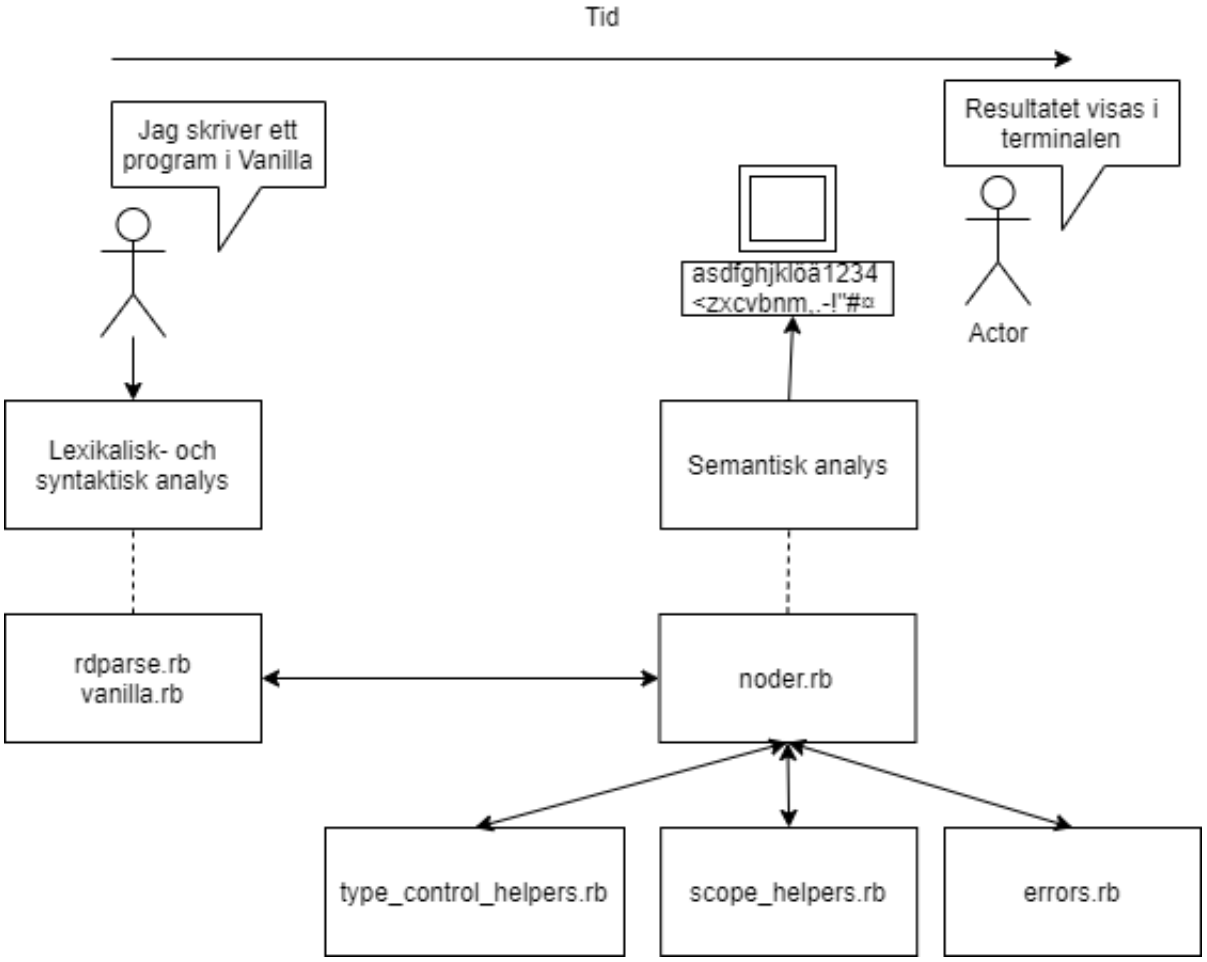
```
1  int chief():
2      list<list<int>> x = [[1, 2], [3, 4]]
3      int res
4      int n = 4 # 'n' deklareras här i det yttersta
       scopet
5
6      for list<int> l in x:
7          int res
8          printn(n) #Kan nå 'n'
9
10         for int num in l:
11             printn(n) #Kan också nå 'n'
12             res += num #'res' i den första for-each
           loopen ändras
13         endfor
14
15     endfor
16
17     int z = 2
18     int w = scope_test(n)
19
20     return res
21
22 end
23
24 int scope_test(int n):
25     return n + z #Kastar DeclarationError, z kan inte
       nås från denna funktion.
26 end
27
28 #chief returnerar 0 eftersom 'res' i det yttersta
       scopet inte har ändrats
```

Programexempel 29: scope.

3 Systemdokumentation

RDparse används för att utföra lexikalisk- och syntaktisk analys (parsning) av Vanilla koden. Detta resulterar i ett abstrakt syntax träd. Trädet består av noder i form av klassobjekt som representerar de språkkonstruktioner som finns i språket. Dessa evalueras sedan i den semantiska analysen genom objektens evaluate funktion. Koden för RDparse finns i filen rdparse.rb.

3.1 Översikt



3.2 Lexikalisk analys

Den lexikaliska analysen skapar tokens av koden, dvs. nyckelord som ska sparas för vidare analys. Reguljära uttryck beskriver vilka delar av koden som ska bilda tokens i Vanilla. Bilden nedan visar exempel på hur Int och Float matchas. Lexern matchar det reguljära uttrycket och hanterar matchingen på det sätt som anges i det tillhörande blocket. Sedan sparas resultatet i en array som senare används i parsningen. Koden för alla tokens finns i filen vanilla.rb.

```
token(/\d+\.\d+/) {| float | float.to_f() }
token(/\d+/) { | integer | integer.to_i() }
```

Exempel på token-matchningar

3.3 Parsning

Parsern använder sig utav de tokens som kommer från den lexikaliska analysen för att parsa koden enligt de syntaktiska regler som finns. Varje regel innehåller en beskrivning av vad som ska matchas samt ett block som anger vad som ska göras med matchningnarna. Reglerna utgår från språkets grammatik, se bnf:en i rubrik 3.4. Med hjälp av dessa regler skapar parsern ett abstrakt syntaxträd.

Bilden nedan visar hur regeln ser ut för funktionsanrop. Parsern skapar ett objekt av klassen FunctionCall enligt blocket som sedan ingår i det abstrakta syntaxträdet. Koden för reglerna finns i filen vanilla.rb.

```
rule :function_call do
    match(:name, :args) {|name, args|
    FunctionCall.new(name, args)}
end
```

Exempel på parsnings-regel

3.4 Grammatiken för språket

<program>::= <separator><function><separator><program><separator>
|<function><separator><program><separator>
|<separator><function><separator><program>
|<function><separator><program>
|<separator><function><separator>
|<function><separator>
|<separator><function>
|<function>
|<separator>

<function>::= <types><name><parameters>:<block>end

<parameters>::= ( )
|(<parameter_list>)

<parameter_list>::= <declaration>,<parameter_list>
|<declaration>

<block>::= <separator><valid><block>
|<separator><valid><separator>
|<separator>

<separator>::= [\n\r]+ <separataor>
|[\n \r]+

<valid>::= <statement>
|<lambda_expression>

<statement>::= <declaration>
|<assignment>
|<return>
|break
|next
|<print>
|<if>
|<iteration>

<declaration>::= <types><name>=<lambda_expression>
|<types><name>

<assignment>::= <name><assign_op><lambda_expression>

<assign_op>::= =
|+=
|-=
|*=
|/=
|^=
|%=

<return>::= return<lambda_expression>

<print>::= print(<lambda_expression>)
|printn(<lambda_expresison>)

<if>::= <simple_if>
|<compounded_if>

<simple_if>::= if <lambda_expression>:<block>endif

<compounded_if ::= if <lambda_expression>:<block><branches>endif

<branches>::= <elseif_branch><branches>
|<elseif_branch>
|<else_branch>

<elseif_branch>::= elseif <lambda_expression>:<block>

<else_branch>::= else :<block>

<iteration>::= <for_loop>
|<for_each>
|<while_loop>

<for_loop>::= for <declaration>, <lambda_expression>,
<assignment >:<block>endfor

<for_each>::= for <declaration>in <lambda_expression>:
<block>endfor

|for <declaration>, <declaration>in <lambda_expression>
: <block>endfor

|for <auto>in <lambda_expression>:
<block>endfor

|for <auto>, <auto>in <lambda_expression>:
<block>endfor

<auto>::= auto <name>

<while_loop>::= while <lambda_expression>: <block>endwhile

<lambda_expression>::= [ ] <parameters>: end
|[<captures>] <parameters>: end
|[] <parameters>: <lambda_expression>end
|[<captures>] <parameters>: <lambda_expression>end
|<bool_expression>

<captures>::= <name>, <captures>
|<name>

<bool_expression>::= <bool_expression>or<bool_term>
|<bool_term>

<bool_term>::= <bool_term>and<bool_factor>
|<bool_factor>

<bool_factor>::= not<bool_factor>
|!<bool_factor>
|<comp_expression>

<comp_expression>::= <comp_expression><comp_operator><artihmetic_expression>
|<artihmetic_expression>

<comp_operator>::= >
|<
|>=
|<=
|==
|!=

<arithmetic_expression>::= <arithmetic_expression>+<arithmetic_term>
|<arithmetic_expression>-<arithmetic_term>
|<arithmetic_term>

<arithmetic_term>::= <arithmetic_term>*<arithmetic_factor>
|<arithmetic_term>/<arithmetic_factor>
|<arithmetic_term>%<arithmetic_factor>
|<arithmetic_factor>

<arithmetic_factor>::= <method>^<arithmetic_factor>
|<method>

<method>::= <method>.<name><args>
|<method>[<lambda_expression>] =<lambda_expression>
|<method>[<lambda_expression>]
|<atom>

<atom>::= <input>
|<function_call>
|"-" Float
|"-" Integer
|Float
|Integer
|VString #vanilla string node
|<list>
|<map>
|true
|false
|<name>
|(<lambda_expression>)

<function_call>::= <name><args>

<input>::= input (<lambda_expression>)
|input()

<list>::= [<arg_list>]
|[]

<map>::= {<pairs>}
|{}

<pairs>::= <key_value>,<pairs>
|<key_value>

<key_value>::= <lambda_expression>:<lambda_expression>

<args>::= ()
|(<arg_list>)

<arg_list>::= <lambda_expression>,<arg_list>
|<lambda_expression>

<types>::= map<<types>,<types>>
|list<<types>>
|<type>

<type>::= int
|string
|float
|nothing
|bool
|lambda

<name>::=^(?!\[$)(?!\]$)(?!printn$)(?!print$)(?!input$)(?!end$)(?!endwhile$)(?!endfor$)(?!endif$)(?!do$)
[a-z_][0-9a-zA-Z=_]*[!?]?$

3.5 Abstrakt Syntaxträd

Det abstrakta syntaxträdet består av noder i form av klassobjekt som representerar de språkkonstruktioner som finns i språket. Den första noden är en instans av klassen Program, vilket består av en array av alla funktioner som finns i programmet. Funktionerna i sin tur innehåller objekt som representerar andra delar av programmet till exempel for-loop och deklarering. Alla klasser i vanilla har en funktion som heter evaluate. evaluate anropas genom den första noden i trädet(program), och sedan anropar den evaluate i programmets huvudfunktion som heter chief vilket resulterar i att resterande objekt evalueras. Till sist returneras resultatet från chief. Koden för alla nodklasser finns i filen nodes.rb.

3.6 Evaluate

Grundstrukturen i programmet är som nämnt ovan, varje del representeras av en instans av respektive klass och under körning anropas objektens evaluate

funktion. I de flesta fallen används en klass för att representera delarna men det finns undantag där två olika klasser är implementerade. Detta gäller konstruktionerna for-loop, for-each och deklarering.

Dessa delar har en definitionsklass som lagrar information samt en klass som sköter evaluering av innehållet. När evaluate anropas för defintitionsklassen skapar denna en instans av evaluate noden och anropar evaluate via den noden. Detta implementerades eftersom dessa konstruktioner har information som ändras under evalueringen vilket innebär att samma objekt inte kan användas flera gånger i rekursion.

3.7 Typkontroll

Vanilla är ett typat språk och därför är typkontroll implementerat. Om fel typ påträffats kastas ett felmeddelande som anger vilken typ som objektet har samt vilken typ som förväntades. För att hantera detta finns det olika hjälpfunktioner.

Huvudfunktionen för typkontrollen heter type_control och denna kontrollerar så att värdet är av rätt typ samt anropar funktionerna assert_types_map eller assert_types_array om värdet är en hash eller array. I dessa fall måste även elementen eller key/value par typkontrolleras. Funktionerna get_type och has_type tillkommer för att underlätta jämförelser av typer i de andra funktionerna.

Typkontroll genomförs vid deklarering och tilldelning av variabler, vissa metodanrop samt då sammansatta datatyper har skrivits ”fritt” av användaren, till exempel om en lista skrivs ut i terminalen på formatet: print([1, 2, 3]).

Vid deklarering och tilldelning har respektive nodklass information om vad som är korrekt datatyp(er) vilket är ett resultat från parsningen. När data har skrivits fritt finns däremot ingen information om vilken typ det är/innehåller. Därför finns det en funktion som heter make_type_structure. Denna returnerar information om vilken datatyp ett objekt är/består av vilket sedan anges som argument till funktionen type_control. I make_type_structure är det första elementet/key-value paret som får avgöra vad som är korrekt datatyp för en List eller Map.

Efter metodanrop som ändrat på en sammansatt datastruktur utförs typkontroll på det nya värdet. Om objektet metoden anropades genom var lagrad i en variabel hämtas information om korrekt datatyp från variabel-noden som är lagrad i $variables, annars används funktionen make_type_structure som förklarades ovan. Observera att det enbart är det nya värdet som kontrolleras, inte hela datastrukturen.

Mer information om de olika funktionerna finns i filen type_control_helpers.rb.

3.8 Scopehantering

Scope hanteras med hjälp av en global array som heter $variables där varje index innehåller en hash och varje hash representerar scopet i en av programmets funktioner. Vid evaluering av if-satser och repetitionssatser skapas en ny hash med en :parent nyckel via funktionen add_parent_scope. :parent-scopet innehåller information om scopet som satsen är skriven i, dvs. det tidigare scopet lagrat i arrayen.

När en variabel ska nås används hjälpfunktionen look_up som letar i den yttersta hashen först innan hashar som :parent-nycklar leder till sökes igenom. På så vis är det alltid variabler i det närmaste scopet som nås först. Samma logik gäller vid tilldelning av variabler, om det finns variabler med samma namn i olika scope är det variabeln som är mest lokal som får det nya värdet. Vid deklarering av nya variabler placeras dessa i den yttersta hashen, alltså i nuvarande scope-nivå. Efter evaluering av satsen återställs scopet genom att scope blir vad :parent innehåller igen via funktionen pop_parent_scope.

Vid rekursion innehåller hashen i $variables alltid det scope som gäller för den nuvarande nivån i rekursionen. Scope från tidigare nivåer har sparats i en annan array som heter $stack. När ett rekursivt funktionsanrop har nått basfallet ändras scopet i $variables till den tidigare nivån vilket är det sist tillagda på stacken, samtidigt raderas detta från stacken. På så vis töms stacken med ett scope i taget.

När lambda-funktioner anropas tillges funktionskroppen ett nytt tomt scope, samtidigt som nuvarande scope i $variables läggs på stacken som vid ett rekursivt anrop. I det tomma scopet läggs parametrarna samt eventulla variabler i captures till.

$variables och $stack samt funktionen look_up, add_parent_scope och pop_parent_scope finns i filen scope_helpers.rb. De är placerade där eftersom flera klasser har metoder som använder dem. VariableDeclarationEval har en funktion för deklarering eftersom det bara är i den noden som det sker. Funktionen 'assign' för tilldelning finns dock i scope_helpers.rb eftersom denna används av flera noder.

3.9 Kodstandard

När Vanilla skrevs användes en kodstandard som följer indentering och snake case (dvs. sammansatta ord skiljs åt med _). Klassnamn är skrivna med camel case eftersom det är så rubys syntax ser ut. Vi har genomgående försökt namnge variabler så att de blir så beskrivande som möjligt samt använt paranteser vid funktionsanrop även om Ruby tillåter att de utelämnas.

3.10 Packetering

Koden finns att hämta på https://gitlab.liu.se/lovja643/vanilla. Ladda ned mappen vanilla.zip. Maila mig vid eventuella problem.

4 Reflektioner

Planen från början i projektet var att implementera ett typat språk som skulle ha listor, tuples, maps samt klara av rekursion. I slutändan valde jag att hoppa över tuples och implementerade istället lambda-funktioner eftersom jag ville göra något nytt istället för en till datastruktur.

Jag har lärt mig mycket under projektets gång och stött på flera situationer där det varit svårare att fatta beslut angående hur en del av språket ska implementeras samt situationer där problemen helt enkelt varit svårare att lösa. Jag tänkte gå igenom detta samt avsluta med vad jag jag lärt mig genom att göra projektet.

Lagring i variabler
Under större delen av projektet lagrades inte ruby's datatyper i variabler utan våra egna datatyper. Vi tyckte att det kändes logiskt att eftersom det inte är ruby utan vårt språk och då skulle även våra datatyper lagras om användaren sedan vill nå värdet igen. Precis i samband med den första inlämningen kom jag dock fram till att det var onödigt för funktionaliteten i programmet och sen dess har jag inte saknat det.

Många beslut innan togs baserat på att det hela tiden skulle vara våra datatyper som skulle lagras, vilket t.ex. gjorde att implementeringen av listor tog för lång tid. Det jag funderade på då var om listornas evaluate-funktion behövde returnera en ruby Array. Vid tillfället kunde jag komma på fler situationer då en ruby array inte var nödvändig än nödvändig varav omvandlingen från vår datatyp List till Array fram och tillbaka kändes onödig. I slutändan är detta inget problem eftersom det inte behövs konvertera tillbaka till en List igen.

Element i listor
En annan fråga angående listor var huruvida uttryck och däribland variabler skulle lagras i listan eller om dessa skulle evalueras. Om uttryck tilläts skulle innehållet i listan förändras om variablerna utanför listan förändrades. I det andra fallet blir istället listans element en kopia av en variabels värde. Till en början tilläts uttryck lagras i listan men jag ändrade på det eftersom det kändes enklare att låta uttrycken evalueras, både att implementera och för användaren.

Det finns dock en situation då detta skapar problem. Om listor eller maps skapas inom repetitionssatser som innehåller iterator-variabeln/variablerna måste uttrycket få står kvar, annars kommer listans element vara det som iteratorvari-

abeln var under första varvet i loopen. Jag har testat olika lösningar för detta, i nuläget görs en djup kopia av arrayen som innehåller allt som står skrivet i ett block så att det är samma "ursprungslista" som evalueras varje varv.

En lärdom från detta till nästa projekt är att tänka igenom implementationer noggrannare innan de påbörjas. I detta fall hade vi inte bestämt oss för hur elementen skulle lagras i listorna men på något sätt så blev det så att listorna innehöll uttryck och variabler som inte var evaluerade. Detta resulterade i ett gäng problem som löstes med olika tillfälliga lösningar för att få det att fungera innan jag började tänka på hur vi egentligen skulle ha det. Om jag hade tänkt igenom det bättre innan hade jag ju kunnat göra som det skulle vara från början!

Rekursion
Rekursion var även en del av programmet som krävde lite mer tankekraft än de grundläggande delarna. Detta berodde dock på hur andra delar av programmet var implementerat snarare än just hanteringen av scope vid rekursion. En sak som jag inte alls tänkt på från början var att när samma klassobjekt körs i ett rekursivt anrop, får det effekt på andra nivåer av rekursionen. Jag satt en bra stund och letade efter orsaken till varför en for-loop aldrig ville avslutas i en rekursiv funktion tills jag insåg att iterator-variabeln på den första nivån alltid nollställdes på en annan nivå och därmed blev aldrig den angivna jämförelsen falsk.

En annan sak som upptäcktes i samband med rekursion var hur ineffektiv typkontrollen var. Typen kontrollerades nämligen innan ett objekt evaluerades för att få ett resultat. För att kontrollera en typ måste objektet också evalueras varav evaluering skedde onödigt många gånger, speciellt i samband med rekursion. Därför var rekursionen i vissa fall väldigt långsam till en början. Efter att jag ändrade på det går det rimligt snabbt för att vara implementerat i Ruby tror jag. Det är oklart varför vi valde den första implementeringen av typkontroll, kanske för att det kändes logiskt att först kontrollera så att "man får göra något" innan det görs.

Typkontroll
Typkontroll är något som funnits i olika versioner. Det nämndes ovan att vi först tänkte fel angående ordningen av typkontrollen vilket resulterade i mycket ineffektivitet. Till en början hade vi också missat att när listor skapas med element som är listor bör typen även anges för elementen i den inre listan. Det var tur att vår handledare upptäckte det under den första inlämningen. Efter detta har typkontrollen funnits i två versioner. I det första fallet typkontrollerades sammansatta datastrukturer först då själva listan/mapen evaluerades. Då fick första elementet/key-value paret avgöra vad som var korrekt typ. Om listan eller mapen sedan skulle lagras i en variabel utfördes även en kontroll gentemot den specificerade typ som angetts för variabeln. Anledningen till denna dubbel-kontroll var för att sammansatta datastrukturer kunde skrivas fritt i koden varav typkontroll krävdes utöver kontrollen i deklarering och tilldelning.

Jag ändrade på den första versionen eftersom det ibland resulterade i konstiga felmeddelanden. Till exempel resulterade följande; list <int >x = [”hej”, 2, 3] i ett felmeddelande som sa att rätt datatyp var String (första elementet) även om det är specificerat att listan ska innehålla Int. Det slutliga versionen utför inte typkontroll under evaluering av listan/mappen. För att kontrollera att en lista/map som skrivs fritt innehåller korrekt datatyper utförs typkontroll på de ställen det krävs, vilket i praktiken var färre ställen än jag tänkt från början. Därmed tror jag att jag undvek en del onödiga typkontroller.

Vad jag har lärt mig
Genom att göra projektet har jag lärt mig mycket om hur ett datorspråk kan byggas, vad en lexer är, vad en parser är och hur en interpretator kan fungera. Innan kurserna TDP007 och TDP019 startade hade jag absolut ingen aning om något av detta.

Om denna reflektion läses noggrant kan du se att jag sköt mig själv i foten några gånger under projektet. Mycket har berott på att jag/vi inte tänkt igenom implementering av olika saker i språket tillräckligt noggrant vilket resulterade i att jag skrev om delar av koden ett antal gånger. Något som också påverkat detta är att jag antagit att tidigare beslut är givna och sedan rättat mig efter dem när senare lösningar implementerats istället för att ha i bakhuvudet att de tidigare lösningarna kanske inte är så fantastiska trots allt. Jag blev bättre på detta i slutändan av projektet efter att jag gjort misstaget några gånger. På så vis upptäckte jag andra delar av koden som jag tror var onödigt komplicerade och kunde göra om dem. Hur som helst bör jag komma ihåg till nästa projekt att se till så att det jag kodar är mer genomtänkt.

En annan viktig lärdom till nästa projekt som görs är nog att se till att testa en delkomponent ordentligt innan nästa sak påbörjas. Det handlar mest om listorna i detta fall som inte var testade så mycket. När jag sedan försökte använda listorna tillsammans med andra delar av språket stötte jag på massa problem som löstes dåligt genom tillfälliga lösningar på de ställen i koden som problemet uppstod. När jag sedan satte mig och testade listorna och metoderna genomgående kunde dessa nödlösningar tas bort. Mycket tid gick alltså åt till att skapa lösningar som hade kunnat undvikas om jag bara att gått igenom List klassen först.

Avslutning
Det har varit kul att göra språket och det känns som att jag förstår programmeringsspråk bättre i efterhand. Eftersom deadline blev förlängt hann jag också lägga till fler konstruktioner i språket samt rätta till en del delar av koden som inte var optimalt. Det finns säkert förbättringspotential men jag har gjort vad jag kan och allt som allt är jag därför nöjd med slutresultatet.