0.1. Projekt: Implementation av ett datorspråk

Författare och utvecklare
Erik Edling, eried975@student.liu.se
Emil Östergren, emios530@student.liu.se

Vårterminen 2015

Innehållsförteckning

1. Inledning

Detta projekt är skapat av studenter som läser TDP019 under termin två på IP-programmet vid Linköpings Universitet. Dokumentet innehåller 4 olika delar, användarhandledning, systemdokumentation, fullständig programkod och reflektioner.

1.1. Introduktion

Resultet av projektet är språket niLANG som är ett svenskt, delvis objekt-orienterat språk då det innehåller klasser. Varför det inte är ett objekt-orienterat språk beror på att allt inte är uppbyggt av dessa. Det var även tänkt att språket skulle innehålla funktionalitet som gjorde det enkelt och smidigt att utföra vissa operationer, exempelvis for-loopen som enkelt kan manipuleras såsom användaren själv vill ha den.

1.2. Syfte

Projektet gick ut på att lära oss att skapa ett eget programspråk. Språket skapades utefter studenternas egna specifikationer.

1.3. Målgrupp

niLANG skapades med målsättningen att vara ett konsekvent språk som kan användas av nybörjare som redan förstår sig på grunderna till programmering. Språket är tänkt att vara väldigt grundläggande men innehåller även mer komplicerade uttryck som till exempel klasser. Därmed kan nybörjare även i lagom nivå lära sig baserna i objekt-orienterad programmering och inte bara imperativ programmering.

2. Användarhandledning

niLANG är ett språk som innehåller en mängd olika funktionaliteter. I denna handledning beskrivs all funktionalitet som ingår i språket. Språkets funktionaliteter innefattar variabler och behållare, aritmetiska uttryck, jämförelser, funktioner, iteratorer, klasser, in- och ut-matning samt villkor. Vad dessa är och hur dom fungerar beskrivs nedan.

2.1. Grundläggande Syntax

niLANGs syntax är likt många andra språk. Språket använder separatorer precis som i C++ och sats-slut som i ruby. i 2.1.1. och 2.1.2. beskrivs det vidare hur dessa fungerar.

2.1.1. Seperatorer

I niLANG används separatorer för att skilja mellan uttryck. Utan att använda sig utav en separator kan språket inte skilja mellan två olika uttryck även om de står på två olika rader. En separator skrivs med tecknet “ ; “(semikolon).

2.1.2. Satsslut

I niLANG finns det många typer satser: för_varje-satser, medans-satser, klass-satser, funktions-satser samt om-satser. Alla dessa behöver avslutas för att språket ska veta när en sats avslutas. Ett satsslut skrivs med ordet “slut”.

2.1.3. Kommentarer

Kommentering i niLANG sker mellan två stycken specialtecken och detta kan göras över flera rader. Specialtecknet som används är “ ¤ “. Ett exempel på detta visas i *Figur 1.*

`het a = 5; ¤ detta är en kommentar!`
`str a = “hej”; detta är fortfarande samma kommentar!¤`
`kskriv a; // 5 skrivs ut i terminalen.`
	Ex1

Figur 1.

2.2. Variabler och behållare

2.2.1. Datatyper

niLANGs variabler skapas med en given typ men är inte begränsad till denna. Detta är på grund utav läsbarhet men även att det ska vara smidigare att kunna använda variablerna. Språket har fyra bastyper, str(strängar), het(heltal), flt(flyttal) och boo(booleaner). Variabler kan förutom dessa typer även vara en behållare eller en klass instans vilket senare tas upp under kapitel 2.3 respektive 2.9.

2.2.2. Tilldelning och deklarering

i niLANG behöver man ange vilket typ en variabel ska vara vid en deklarering. Ett exempel på hur en deklarering fungerar visas i *Figur 2 Ex1.*

`het minvariabel = 5;  //minvariabel tilldelas värdet 5.`
`boo minbool = sant;  //minbool tilldelas värdet sant.`
	Ex1

`het minandravariabel = minvariabel + 5;` `//minandravariabel tilldelas 10.`	**Ex2**
`minvariabel = "en sträng";` `//minvariabel tilldelas "en sträng".`	**Ex3**

Figur 2.

I Ex2 kan vi se ett exempel på en tilldelning av ett aritmetiskt uttryck. Något som kan noteras i Ex3 är att "minvariabel" tilldelas en annan slags datatyp än vad den skapades som. Detta är i niLANG möjligt då typen som anges vi deklaration endast är till för läsbarhet. Det går därför enkelt att använda olika datatyper i en och samma variabel.

2.3. Behållare

Behållare i niLANG är tänkt att vara smidiga att använda då den kan innehålla olika datatyper samtidigt. Det finns även förbyggda funktioner för behållaren. Dessa funktioner är följande: lägg_till, ta_bort, ändra, hämta och storlek. Alla dessa kan ta in olika argument som påverkar hur behållaren blir manipulerad. I *Figur 3* visas olika användningar av dessa funktioner.

`beh minbehållare = [1,”hej”];`	**Ex1**
`minbehållare.lägg_till(1,2,[3,2]);` `//minbehållare innehåller [1,”hej”,1,2,[3,2]]`	**Ex2**
`minbehållare.ta_bort(1 till 4);` `//minbehållare innehåller [1]`	**Ex3**
`minbehållare.lägg_till(0 blir 7,7);` `//minbehållare innehåller [7,7,1]`	**Ex4**
`minbehållare.lägg_till(0 blir 7,7);` `//minbehållare innehåller [7,7,1]`	**Ex5**
`minbehållare.ta_bort(2); //minbehållare innehåller [7,7]`	**Ex6**
`minbehållare.ändra(1 blir 5);` `//minbehållare innehåller [7,5]`	**Ex7**
`minbehållare.hämta(1); //Värdet 5 hämtas`	**Ex8**
`minbehållare.storlek(); //Värdet 2 returneras`	**Ex9**

Figur 3.

En sak som bör noteras i *Figur 3* är att vissa utav argumenten till funktionerna är index och inte värden. Ett exempel är Ex3 där det står “1 till 4” vilket menas index 1,2,3 och 4 i behållaren. Ett annat exempel är Ex4 där det står “0 blir 7,7” vilket menas att värderna 7 följt av 7 läggs till från och med index 0.

Det finns även två andra funktioner som är värda att nämna, dessa är funktionerna “hämta” och “storlek”. “hämta” kan användas för att få ut ett värde på ett specifikt index i behållaren. “storlek” används för att få ut antalet element i behållaren.

2.4. Aritmetiska uttryck

I niLANG kan relativt avancerade aritmetiska uttryck skrivas med hjälp av ett antal olika operatorer. Operatorerna som är tillgängliga i språket är +, -, *, /,och ^. I ett aritmetiskt uttryck fungerar även parentes mycket bra för att beskriva prioritet. I niLANG är operatorerna tänkta att vara kraftfulla, det vill säga att man kan multiplicera och addera strängar, heltal, flyttal, booleaner och behållare, inom logiska gränser. Det går till exempel inte att multiplicera en sträng och en behållare.

`5+5*(2+2); //Resultatet blir 25`	**Ex1**
`"hej" + 2 + sant + [1,2] + 2.2` `//Resultatet blir strängen "hej2sant[1,2]2.2"`	**Ex2**
`"hej" * 2; //Resultatet blir "hejhej"`	**Ex3**
`[1,2] * 2; //Resultatet blir [1,2,1,2]`	**Ex4**
`het a = 5;` `het b = a+2; //B tilldelas 7`	**Ex5**

Figur 4.

I Figur 4 kan vi se basen av de aritmetiska uttryck som kan skrivas i niLANG. Exemplen visar endast några simpla uttryck men mer komplexa uttryck går självklart att skriva. Vad som kan noteras är att det inte har någon betydelse vilka datatyper vi använder.

2.5. Jämförelse uttryck

En jämförelse i niLANG går till på ett väldigt liknande sätt som i andra programmeringsspråk. Jämförelsen jämförs med hjälp utav relationsoperatorer. Dessa operatorer är “>”, “<”, “>=”, “<=”, “==” och “!=” (allt från variabler till direkt skriva uttryck) . Exempel av detta kan ses i Ex1 i *Figur 5.*

`het tal = 5;`
`tal < 6; // Resultatet blir sant`
`tal > 3+4-2 // Resultatet blir falskt`
	Ex1

`tal <= 4 eller tal >= 5; // Resultatet blir sant`
`tal == 5 och tal > 6 // Resultatet blir falskt`
	Ex2

Figur 5

I niLANG används jämförelse uttryck för att bestämma ifall något ska hända eller hur länge något ska hända. Exempelvis används det för att beskriva ifall en om-sats (förklaras i 2.11.) ska utföras.
I Ex2 i *Figur 5* visas det hur det även som i andra språk går att koppla samman flera jämförelse uttryck med hjälp av de logiska operatorerna “och”, “&&”, “||” samt “eller”.

2.6. Räckvidd

En väldigt viktig funktion att hålla koll på i niLANG är räckvidden. Räckvidden gör så att variabler som skapas inuti en sats endast har en livstid lika lång som själva satsen. Det vill säga att om det skapas en variabel i en funktion(funktioner förklaras vidare i 2.7) går det nu att komma åt variabeln i funktionen. Dock när man har avslutat funktionen försvinner variabeln.

Ex1

```
het b = 2;
definiera funktion minfunktion()
  het b = 3;
  b += 2;
slut

minfunktion(); //b kommer fortfarande vara 2
```

Figur 6.

I *Figur 6* kan vi observera en väldigt viktig sak i niLANG. Om man skapar en variabel utanför en sats och sedan skapar en variabel med samma namn i en sats kommer den variabeln bete sig som en ny variabel, det vill säga att om man försöker manipulera variabeln kommer det endast ske inuti funktionen. Variabeln utanför funktionen kommer inte påverkas alls.

2.7. Funktioner

En funktionalitet i niLANG är att det går att skapa egna funktioner. Dessa dessa funktioner behöver inte som i många andra språk returnera en specifik datatyp, detta eftersom niLANG inte har ett starkt typat språk. Detta medför att en funktion kan returnera en variabel med en valfri datatyp. För att returnera en variabel skriver man helt enkelt bara “returnera” och sedan variabeln man vill returnera.

Funktioner kan skapas med parametrar, dessa parametrar kan inte ha ett initieringsvärde. Detta betyder att en kallelse på en funktion med 2 parametrar måste ha 2 argument i sig vilket visas i Ex1 i *Figur 7*. Självklart behöver man inte några parametrar vilket visas i Ex2.

Ex1

```
definiera funktion minfunktion(het a, str b)
  returnera a+1;
slut

het b = minfunktion(2, “hej”) // B tilldelas 3
```

Ex2

```
het b = 2;
definiera funktion minfunktion()
  het b = 5;
slut

minfunktion(); //b kommer fortfarande vara 2
```

Figur 7.

2.8. Iteratorer

En intressant fakta om niLANG är att även fast det endast finns en iterator så är denna väldigt kraftfull. “för varje” iteratorn är byggd för att enkelt manipulera strängar och språkets behållare. Som det går att se i Ex1 och Ex2 i *Figur 8* måste endast typen som ska konverteras till ändras för att välja ifall iteratorn ska gå över varje bokstav eller varje ord. Som det går att se i Ex3 så går det även att använda datatypen “het” för att ta ut varje nummer ur en “korrekt” skriven sträng, vilket även fungerar med flyttal (flt). I Ex4 i *Figur x* går det även att se hur enkelt det är att iterera genom varje element i en behållare.

`str minsträng = “en sträng med ord”`	

`för varje(minsträng till bok minbokstav) //Skriver ut “e”,”n”,” “,”s”,”t”,...`
`  kskriv minbokstav;`
`slut`
	Ex1

`för varje(minsträng till str minbokstav) //Skriver ut “en”, “sträng”,...`
`  kskriv minbokstav;`
`slut`
	Ex2

`minsträng = “11 222 4”`
`för varje(minsträng till het mittnummer) //Skriver ut 11, 22, 4`
`  kskriv mittnummer;`
`slut`
	Ex3

`beh minbehållare = [1,2,3,4,5];`
`för varje(minbehållare till mittelement) //Skriver ut 1,2,3,...`
`  kskriv mittelement;`
`slut`
	Ex4

Figur 8

2.9. Klasser

niLANG har implementerat de mest basiska begreppen för klasser. Det går att skapa klasser, skriva en initieringsfunktion samt flera andra medlemsfunktioner till dessa och skapa instanser av klassen. I niLANG är variabler i klasser privata och funktioner publika vilket låter användare vänja sig med idéen om dessa begrepp utan att behöva oroa sig om att implementera dem. Ifall användaren vill manipulera variabler direkt från en klassinstans så måste “getters” och “setters” funktioner skapas. Exempel på en klassdefinition, klassinstans deklaration och användning kan ses i *Figur 9*.

definiera klass minklass
//Skapar klassen med variabel siffra och funktionen geSiffra
  definiera initiering(het num)
  het siffra = num+5;
  slut
 	
  definiera funktion geSiffra()
  returnera siffra;
  slut
slut
	Ex1

minklass klassinstansen(5); //Skapar instans av klassen
kskriv klassinstansen.geSiffra(); //Skriver ut värdet som har returnerats av funktionen
	Ex2

Figur 9.

Något som skiljer sig i niLANG från andra språk är att variabler som tas in i parameterlistan automatiskt sparas ner som klassvariabler.

2.10. In- och ut-matning

Språket innefattar uttryck för att skriva ut data i terminalen samt läsa in användarinmatning. Dessa uttryck är väldigt enkla att använda.

Ett in-matningsuttryck kan manipuleras på olika sätt. Som det går att se i Ex1 i *Figur 10* skrivs inmatningen in till ett heltal vilket fungerar utmärkt om inmatningen från användaren är ett heltal. Detta fungerar för alla datatyper förutom behållare och booleaner.

Ett ut-matningsuttryck är ett väldigt smidigt uttryck i niLANG. Det spelar ingen roll vad för slags datatyp variabeln har som man vill skriva ut. Ut-matninguttrycket kan dessutom ta ett aritmetiskt uttryck som utskrivning.

het a; kläs a; //a tilldelas ett heltal som matas in av användaren	**Ex1**
str b; kläs b; //b tilldelas en sträng som matas in av användaren	**Ex2**
str minsträng = "hej"; kskriv minsträng; //"hej" skrivs ut i terminalen	**Ex3**
str minsträng = "hej"; beh minbehållare = [1,2,3]; kskriv minsträng + minbehållare +(5+5) //"hej[1,2,3]10" skrivs ut i terminalen	**Ex4**

Figur 10.

En viktig sak som bör noteras i Ex4 i *Figur 10* är varför det blir 10 i slutet och inte 55. Eftersom uttrycket i "kskriv" är ett aritmetiskt uttryck används parenteser för prioritet, då vi har parenteser runt "5+5" prioriteras detta och resultatet blir strängen "hej[1,2,3]10".

2.11. Villkor

Villkor i niLANG har stora liknelser till hur de ser ut i andra språk. Språket innefattar två olika villkors-satser, om-satser och medans-satser. En om-sats kan innehålla annars_om-satser samt en annars-sats men det är inget krav. Om-satser kan därmed sätta villkor på vad som ska utföras med hjälp av jämförelseuttryck.

Till skillnad från om-satser så utför medans-satser något upperapade gånger medans ett villkor stämmer. På grund utav detta har satserna två helt olika användningsområden.

Exempel på villkors-satser kan ses i *Figur 11.*

Ex1

```
het a = 2;

om (a < 0)
  kskriv "talet är negativt!"
annars om(a>0)
  kskriv "talet är positivt!"
annars
  kskriv "talet är noll"
slut
```

Ex2

```
het a = 0;

medans (a < 5)
  a += 1;
  kskriv a;
slut
//Resultatet blir 5 utskrivningar 1-5.
```

Figur 11.

3. Systemdokumentation

3.1. Översikt

niLANG använder sig utav en given RDparser. Denna börjar med att göra en lexikalisk analys för att bilda så kallade tokens. Dessa används för parsa kodfilen. Vid parsningen bildas ett abstrakt syntaxträd av alla objekt, sedan är det enkelt att evaluera hela trädet genom en "kedjereaktion".

3.1.1. Lexikalisk analys

Under den lexikaliska analysen bildar niLANG tokens som senare kan användas vid parsningen. Detta sker genom att texten i kodfilen jämförs med reguljära uttryck som hämtar ut valda tokens och behandlar dem. De olika tokens som hämtas ut är:

- ❏ Kommentarer
- ❏ Specialfall för klassdefiniering, klassinitiering samt annars om-sats
- ❏ Strängar
- ❏ Blanksteg
- ❏ Relationsoperatorer och logiska operatorer
- ❏ Flyttal
- ❏ Heltal
- ❏ Variabelnamn
- ❏ Resterande tecken

3.1.2. Parsning

Parsningen går till så att alla de tokens som sparats jämförs med olika regler uppsatta utifrån ett BNF-träd. När en matchning sker returneras en bestämt uppsättning av data. Detta gör att ett abstrakt syntaxträd bildas.

3.1.3. Abstrakt syntaxträx

När hela kodfilen har parsats igenom kommer huvudprogrammet att evaluera rotnoden vilken är av klassen Program_nod. När denna nod evalueras kommer den i sin tur att evaluera alla grenar i trädet. Varje nod i trädet har en funktion som kallas eval(). Detta funktion utför det klassen är byggd för. Finns det fler grenar att evaluera kommer denna funktion i sin tur att kalla på den nodens funktion eval().

3.1.4. Kodstandard

niLANG sätter väldigt få gränser för val av kodstandarder. En sats ska avslutas med “slut” och ett ensamt uttryck avslutas med “ ; “. Det har därför inte någon betydelse vart dessa avslutningar placeras så länge de är i rätt ordning.

Alla variabelnamn, funktionsnamn och klassnamn får bara innehålla bokstäver och inga specialtecken, siffror med mera. Det har ingen betydelse om det är versaler eller gemener.

3.1.5. Grammatik

Språket är byggt utefter BNF-grammatik. Nedan finner du BNF-grammatiken för niLANG.

```
<program>::= <flera_satser>

<flera_satser>::= <flera_satser> <sats>
                | <sats>

<sats>::=  <behållare_uttryck> “;”
                | <konsol_uttryck> “;”
                | <tilldelnings_uttryck> ”;”
                | <om_block>
                | <förvarje_sats>
                | <medans_sats>
                | <funktions_definition>
                | <klass_definition>
                | <klass_instans> “;”
                | <jämförelse_uttryck> “;”
                | <tomt>

<behållare_uttryck>::= “beh” <sett_variabel> “=” “[“ argument_lista “]”
                | “beh” <sett_variabel>
                | <sett_variabel> “.” “lägg_till” “(“ <argument_lista> “)”
                | <sett_variabel> “.” “lägg_till” “(“ <Integer> “blir” <argument_lista> “)”
                |<sett_variabel> “.” “ta_bort” “(“ <Integer> “till” <Integer> “)”
                | <sett_variabel> “.” “ta_bort” “(“ <Integer> “)”
                | <sett_variabel> “.” “ta_bort” “(“ “)”
                | <sett_variabel> “.” “ändra” “(“ <Integer> “blir” <aritmetiskt_uttryck> “)”

<behållare_egenskap_uttryck>::= <sett_variabel> “.” “hämta” “(“ <Integer> “)”
                | <sett_variabel> “.” “storlek” “(“ “)”

<tilldelnings_uttryck>::= <sett_variabel> “=” <aritmetiskt_uttryck>
                | <sett_variabel> (“+=” | “-=” | “*=” )  <aritmetiskt_uttryck>
                | <namn> <sett_variabel> “=”  <aritmetiskt_uttryck>
                | <namn> <sett_variabel>

<om_block>::= <om_sats>

<om_sats>::= <om> <annars_om> <annars> “slut”
                | <om> <annars_om> “slut”
                | <om> <annars> “slut”
                | <om> “slut”

<om>::= “om” “(“ <jämförelse_uttryck> “)” <flera_satser>

<annars_om>::= “annars om” “(“ <jämförelse_uttryck> “)” <flera_satser>
                | <annars_om> <annars_om>
```

```
<annars>::= “annars” <flera_satser>

<förvarje_sats>::= “för” “varje” “(“ <sett_variabel> “till” <förvarje_typ> <sett_variabel>“)”
        <flera_satser> “slut”
                | “för” “varje” “(“ <sett_variabel> “till” <sett_variabel> “)” <flera_satser>
                “slut”

<förvarje_typ>::= “bok”
                | “str”
                | “het”
                | “flt”


<medans_sats>::= “medans” “(“ <jämförelse_uttryck> “)” <flera_satser> “slut”

<konsol_uttryck>::= “kskriv” <aritmetiskt_uttryck>
                | “kläs” <sett_variabel>


<jämförelse_uttryck>::= <eller_uttryck>

<eller_uttryck>::=<eller_uttryck> ( “eller” | “||” ) <och_uttryck>
                | <och_uttryck>

<och_uttryck>::= <aritmetiskt_uttryck> <bool_operator> <aritmetiskt_uttryck>
                | <och_uttryck> ( “och” | “&&” ) <jämförelse-uttryck>
                | <boolean>
                | <aritmetiskt_uttryck>

<funktions_definition>::= “definierar” ”funktion” <namn> “(“ <parameter_lista> “)”
        <funktions_satser> ‘slut’
                | “definierar” ”funktion” <namn> “(“  “)” <funktions_satser> ‘slut’

<funktions_satser>::= <funktions_satser> <funktions_sats>
                | <funktions_sats>

<funktions_sats> <behållare_uttryck> “;”
                | <retur_uttryck> “;”
                | <konsol_uttryck> “;”
                |<tilldelnings_uttryck> ”;”
                | <om_block>
                | <medans_sats>
                | <jämförelse_uttryck> “;”
                | <klass_instans> “;”
```

```
<funktions_anrop>::= <namn> “(“ <argument_lista> “)”
                | <namn> “(“ “)”

<mel_funktions_definition>::= “definiera” “funktion” <namn> “(“ <parameter_lista> “)”
        <mel_funktions_satser> “slut”
                | “definiera” “funktion” <namn> “(“ “)” <mel_funktions_satser> “slut”

<mel_funktions_satser>::= <mel_funktions_satser> <mel_funktions_sats>
                | <mel_funktions_sats>

<mel_funktions_sats>::= <behållare_uttryck> “;”
                | <retur_uttryck> “;”
                | <konsol_uttryck> “;”
                |<tilldelnings_uttryck> ”;”
                | <om_block>
                | <medans_sats>
                | <jämförelse_uttryck> “;”

<mel_funktions_anrop>::= <sett_variabel> “.” <mel_funktion>

<mel_funktion>::= <namn> “(“ <argument_lista> “)”
                | <namn> “(“ “)”

<klass_definition>::= “definiera klass” <namn> <init_sats> <klass_satser> “slut”
                | “definiera klass” <namn> <klass_satser> “slut”

<klass_satser>::= <klass_satser> <klass_sats>
                | <klass_sats>

<klass_sats>::= <mel_funktions_definition>

<init_sats>::= “definiera initiering” “(“ <parameter_lista> “)” <mel_funktions_satser> “slut”
                | “definiera initiering” “(“ <parameter_lista> “)”  “slut”
                | “definiera initiering” “(“  “)” <klass_satser> “slut”

<klass_instans>::= <namn> <sett_variabel> “(“ <argument_lista> “)”
                | <namn> <sett_variabel>

<aritmetiskt_uttryck>::= <aritmetiskt_uttryck>  (“+” | “-”)  <multi_uttryck>
                | <multi_uttryck>

<multi_uttryck>::= <mult_uttryck> (“*” | “/”) <faktor_uttryck>
                | <faktor_uttryck>
```

```
<faktor_uttryck>::= <prio_uttryck>  “^” <faktor_uttryck>
                  | <prio_uttryck>

<prio_uttryck>::= “(“<aritmetiskt_uttryck> “)”
                  | <fragment>

<fragment>::= “[“ <argument_lista> “]”
                  | <funktions_anrop>
                  | <behållare_egenskap_uttryck>
                  | <mel_funktions_anrop>
                  | <boolean>
                  | regex(\".*?\")
                  | <variabel>
                  | “-” Float
                  | Float
                  | “-” Integer
                  | Integer

<parameter_lista>::=  <parameter_lista> “,” <namn> <sett_variabel>
                  | <namn> <sett_variabel>

<argument_lista>::= <argument_lista>  “,” <aritmetiskt_uttryck>
                  | <aritmetiskt_uttryck>

<retur_uttryck>::= “returnera” <aritmetiskt_uttryck>

<namn>::= <datatyp>
                  | regex([^(slut)][a-zA-ZåäöÅÄÖ]*)

<datatyp>::= “het”
                  | “flt”
                  | “str”
                  | “boo”

<bool_operator>::= “==” | “!=” | “<” | “>” | “<=” | “>=”

<boolean>::= “sant” | “falskt”

<variabel>::= regex([A-Za-zåäöÅÄÖ]+)

<sett_variabel>::= regex([A-Za-zåäöÅÄÖ]+)
```

3.1.6. Installation

1. Se först till att ruby är installerat.
2. Packa upp filen till en godtycklig mapp.
3. Navigera en konsol till den godtyckliga mappen.
4. Skapa en textfil i samma mapp och skriv din önskade kod.
5. Mata in ruby niLANG.rb *filnamn* i konsolen
6. Nu körs din kod

4. Reflektion

Projektet har gått bättre än planerat. Många delar verkade väldigt svåra som till exempel klasser, räckvidd samt funktionsanrop. Dessa var till en början svåra att förstå och fick oss att få tänka till några gånger extra men det gick snabbt framåt. Vid skapning av klassfunktionaliteten spenderade vi en hel dag varpå vi kom till en punkt där det helt enkelt inte var möjligt att fortsätta. Detta var självklart frustrerande men det vi gjorde var att börja om från början och tänka om i nya banor. Vi märkte snabbt att det vi tidigare gjort nu var en genväg mot målet. Vi blev därmed väldigt snabbt klara med klasserna som var den mest avancerade delen av projektet.

Vid början av projektet var det väldigt svårt att förstå hur BNF grammatiken skulle hänga ihop då vi inte visste så mycket om hur implementationsdelen av projektet skulle utföras. Detta medförde att vi flera gånger under projektets gång var tvungna att skriva om små delar av BNF grammatiken så att den gick ihop med implementationen. Ett exempel på detta var när vi skulle implementera klasser för om-satser. Själva syntaxen var från början helt korrekt och det gick att parsa men däremot gick det inte ihop med klassimplementationen av satsen. Vi blev tvungna att skriva om hela om-sats strukturen.

Under projektets gång var det några saker som antingen inte kom med eller nya funktionaliteter vi inte hade tänkt på från projektstarten. Den funktionaliteten som inte kom med var den vanliga for-loopen. Vi valde att inte implementera den på grund utav tidsbrist då vi hade andra viktigare delar kvar att implementera. De funktionaliteter som vi lade till var retur-satser samt behållaren med alla dess inbyggda funktioner. Eftersom vi implementerat såpass mycket funktionalitet i niLANG glömde vi helt enkelt bort dessa standard funktionaliteter.

I slutänden är vi mycket nöjda med vår prestation. Resultatet niLANG uppfyller alla de mål vi själva satt upp för projektet. Vi har skapat ett språk med många olika funktionaliteter, både simpla samt avancerade.