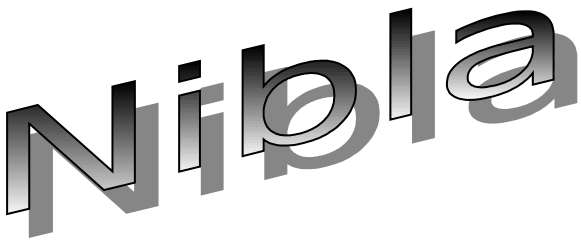
Linköpings universitet
Innovativ Programmering

TDP019 – Datorspråk
VT 2012



Namnet är omvänt men inte koden

Albin Ekberg
albek446@student.liu.se

Innehållsförteckning

1 Inledning

I denna rapport kommer jag ta upp de funktioner som har med Nibla att göra. Hur man använder datorspråket, hur det är uppbyggt och vilka konstruktioner/funktioner datorspråket har. I slutet av dokumentationen kan man läsa mina egna tankar och hur jag gick tillväga för att skapa dataspråket Nibla.

1.1 Bakgrund

Detta är ett projekt gjort av Albin Ekberg som går andra terminen på IP-programmet vid Linköpings universitet. Projektet gjordes för kursen TDP019 - Datorspråk som hölls vårterminen 2012.

Datorspråkets namn är Nibla. Som säkert många ser kan man vända på namnet och då få fram förnamnet på den som skapade detta språket, nämligen Albin. Dock har detta ingen betydelse för när det kommer till vad Nibla är för ett språk.

1.2 Syfte

Syftet med detta datorspråk är att jag ville lära mig hur man kan gå till väga för att skapa ett eget datorspråk. Det är alltså inte meningen att detta skulle bli ett stort datorspråk och användas utav många människor. Det är istället meningen att jag ska få en förståelse för hur andra datorspråk är uppbyggda. Samt att få en förståelse för hur en del av de konstruktioner som används i andra datorspråk ser ut. Därför valde jag att göra ett språk som inte har någon ny eller avancerad syntax för att enklare kunna sätta mig in i hur datorspråk fungerar.

1.3 Kort beskrivning av Nibla

Nibla är ett Imperativt datorspråk som har en syntax som påminner om Python, C++ och Ruby. Dock har Nibla några säregna drag som gör det lite annorlunda än de andra datorspråken. Den största skillnaden är looparna som istället för att använda for- och while-loopar som de flesta datorspråken gör används bara en slags loop. Dock kan man säga att beroende på vad man skriver in i loopen så agerar den på samma sätt som en for- eller while-loop. Sedan är det egentligen bara nyckelorden som skiljer språket från de andra datorspråken. Även om Nibla inte är ett typat språk gällande variablerna är syntaxen ändå ganska strikt. Man kan inte välja att t.ex. skriva med parenteser ibland och utan ibland utan man måste följa den syntax som är skapad för att kunna använda de funktioner man vill. Dock är det enda i syntaxen som är valfritt indentering. Nibla kräver inte att man använder sig utav indentering men det är att rekommendera att använda sig utav det ändå. Allt detta gör att man måste lära sig varje liten del utav språket om man vill lära sig programmera i Nibla.

2 Användarhandledning

I denna användarhandledning kommer läsaren få lära sig grunderna för hur Nibla är uppbyggt. Läsaren kommer också få bättre förståelse för Niblas syntax och konstruktioner.

Nibla är skrivet i Ubuntu så därför rekommenderas att man använder något Linux-system för att undvika problem vid körning. Den här handledningen kommer endast rikta in sig på hur man använder språket i Ubuntu.

2.1 Målgrupp

Nibla är först och främst riktat till personer som har börjat lära sig programmering. Datorspråket är inte svårt att förstå men om man aldrig har programmerat förut kan det vara lite klurigt att förstå allting.

2.2 Starta Nibla

Nibla använder sig utav Ruby1.9 vilket betyder att användaren måste ha Ruby installerat för att kunna använda datorspråket.

För att starta den interaktiva tolken måste man först öppna en terminal och sedan lokalisera den mapp där filerna ligger. När man befinner sig i den mapp där filerna ligger skriver man in kommandot *ruby Nibla_Parser.rb* och trycker på enter. Då kommer Niblas tolk att startas och man kan nu använda datorspråket. För att sedan köra ett program kan man antingen skriva koden direkt i tolken och trycka på enter. Om man har skrivit in rätt kommer språket göra det som användaren har matat in. Man kan också välja att skriva all sin kod i ett dokument som måste ligga tillsammans med de andra filerna för att man ska kunna öppna det. För att köra denna fil öppnar man först tolken i Nibla och skriver sedan följande kommando: *open filnamn* där man ersätter *filnamn* med den fil man vill skicka in. Observera att när man skriver kod direkt i tolken måste allt stå på samma rad medan när man skriver i ett dokument måste man skriva varje sats på ny rad. Skriver man många satser efter varandra inuti en if-sats, en loop eller en funktion måste man avsluta varje sats med ett komma för att på så sätt separera varje sats. För att sedan avsluta Niblas tolk skriver man nyckelordet *bye*.

2.3 Att programmera i Nibla

Man kan köra ett program som är skrivet i Nibla på två olika sätt. Man kan antingen skriva koden direkt i den interaktiva tolken som då kör koden direkt, eller skriva koden i ett dokument med filändelsen *.nibla* och sedan köra hela programmet på en gång.

2.4 Konstruktioner i Nibla

Här visas några exempel på hur man använder vissa funktioner i Nibla.

2.4.1 Datatyper

I Nibla finns det fyra olika datatyper, nämligen:

```
Heltal:       t.ex. 4, 6, 100, 123456
Strängar:     t.ex. ”hej”, ”a string”
Listor:       t.ex. [1,2,3], [5, ”one”], [[1,2], [3,4]]
Dictionary:   t.ex. {”one”=>1, ”two”=>2}
```

2.4.2 Aritmetiska uttryck

I nibla kan man räkna enklare mattematiska uttryck och på så sätt t.ex. kunna skriva ut summan av två tal. Prioriteringen är följande:

```
1 Parantesuttryck:          (1+2+3)
2 Multiplikation/Divition   2*3, 10/5
3 Addition/Subtration       3+4, 6-2
```

Ett exempel: 1+2*3*(6-2)
Om man räknar från vänster till höger blir detta 52 vilket är fel om man jämför med den ”normala” uträkningen. Men eftersom Nibla har de tre prioriteringarna kommer detta istället bli 25 vilket också är rätt.

2.4.3 Utskrift

Om man inte använder sig utav någon utskrift blir det ett ganska tråkigt program eftersom det kommer verka som om ingenting händer. Därför är det en simpel men ändå viktig del utav Nibla. Detta exempel visar på hur man kan skriva ut *”Hello World!”* i terminalen:

```
print ”Hello World!”
```

Man kan också skriva ut många saker efter varandra genom att skriva uttrycken inuti en parantes separerade med ett komma:

```
print (1, 2, ”tre”, ”fyra”, 5)
```

2.4.4 Kommentar

Om man vill kommentera någonting i koden använder man sig utav #. Programmet kommer då ignorera kommentaren och fortsätta med övrig kod om sådan finns:

```
#Detta är en kommentar
```

2.4.5 Variabler

En variabel i Nibla är inte typad så man kan t.ex. först deklarera variabeln som en integer och sedan deklarera samma variabel som en string. För att deklarera en variabel skriver man först variabelnamnet följt utav = och avslutar med ett uttryck:

```
variabel = 10
a = ”ett exempel”
a = 5
b = 3+3              #b kommer få värdet 6
print b              #6 kommer skrivas ut i terminalen
```

Man kan också låta användaren skriva in värdet på en variabel genom att skriva *input* följt utav variabelnamnet. Användaren kan då mata in antingen ett nummer, en sträng eller ett aritmetiskt uttryck:

```
input number          #låter användaren skriva in ett värde som lagras i variabeln number
```

2.4.6 Villkorssats

En if-sats är ett villkor som antingen är sant eller falskt och beroende på vad detta villkor är kommer olika saker att hända. Vi börjar med en simpel if-sats som undersöker om 1=1 och om detta är sant, vilket det också är, så kommer *”true”* skrivas ut i terminalen. För att markera slutet på if-satsen måste man skriva *endif* i slutet. Observera att indentering inte är nödvändigt men det rekommenderas att man alltid använder sig utav det eftersom det blir både lättare för den som skriver koden och den som läser koden. I dessa exempel används endast == operatorn som jämför två uttryck. Dock kan man jämföra med dessa villkor:

- Lika med: x == y
- Inte lika med: x != y
- Större än eller lika med: x >= y
- Mindre än eller lika med: x <= y
- Större än: x > y
- Mindre än: x < y

```
if(1 == 1)
        print ”true”
endif
```

Man kan också göra saker om det första uttrycket skulle vara falskt. I detta exempel så kan man ha hur många *elseif* som man vill men endast en *else*:

```
if(1 == 2)
        print ”false”   #eftersom 1 inte är lika med 2 kommer ”false” inte att skrivas ut
elseif(”a” == ”b”)
        print ”false”   #jämförelse mellan strängar som inte är sant
else                    #eftersom båda föregående villkoren är falska kommer else köras
        print ”true”
endif
```

Man kan också deklarera en variabel och jämföra med den:

```
a = 10
if(a == 10)
        print ”true”    #eftersom a är lika med 10 kommer ”true” skrvas ut i terinalen
endif
```

Man kan också skriva if-satser inuti andra if-satser vilket kallas nästling:

```
if(1 == 1)
        if(2==2)
                print ”true”    #eftersom båda if-satserna är sanna kommer ”true” skrivas ut
        endif
endif
```

2.4.7 Repetitionssats

Looparna i Nibla är lite speciella i jämförelse med andra datorspråk. I andra datorspråk finns det ofta minst två sorts loopar, nämligen en for-loop och en while-loop, men i Nibla finns det bara en loop. Med hjälp av denna enda loop kan man dock skapa några olika loopar. En loop består först utav nyckelordet loop följt utav ett s.k. rangeuttryck, sanningsuttryck eller iteration. Nu till några exempel och det första är en helt vanlig loop som körs ett visst antal varv:

```
loop(i = 0 upto 10)
        print i         #i deklareras och får värdet 0 första varvet och värdet 1 andra varvet
endloop                 #o.s.v. Därför kommer 0 till 10 skrivas ut i terminalen
```

Man kan också vända på denna loop och loopa från ett högt värde till ett lågt med hjälp utav nyckelordet *downto*:

```
loop(i = 10 downto 0)
        print i         #i looparfrån 10 till 0
endloop
```

Nästa exempel är en loop som körs tills sanningsuttrycket visar falskt:

```
loop(1 == 1)
        print ”oändlig loop?”,          #akta så inte en oändlig loop skapas
        break
endloop
```

Det sista exemplet låter användaren iterera igenom en lista eller en sträng:

```
loop(i in ”ITER”)
        print i         #i deklareras och får värdet I varv ett och värdet T varv två o.s.v..
endloop                 #först kommer I skrivas ut, sedan kommer T skrivas ut på ny rad o.s.v
```

2.4.8 Funktioner

Funktioner är ungefär som variabler. När man deklarerar en funktion kommer man kunna anropa den med hjälp utav nyckelordet *call* följt utav funktionsnamnet och eventuella argument. I en funktion kan man använda hela språket på samma sätt som utanför en funktion. Med hjälp av funktioner kan man återutnyttja kod istället för att skriva samma kod många gånger. Dessa exempel

visar två mycket enkla funktioner som endast gör en utskrift. Men som sagt kan man göra mycket mer. Man kan välja att skapa en funktion helt utan parametrar eller en funktion med ett visst antal parametrar:

```
fun testFun
        print ”Skapade en funktion
endfun

call testFun     #anropar funktionen testFun som i detta fallet skriver ut en sträng

fun newTestFun(number1, number2)
        print number1 + number2
endfun

call newTestFun(6, 4)          #anropar funktionen newTestFun som i detta fall tar två
                               #parametrar och skriver ut värdet av 6 + 4 vilket är 10
```

2.4.9 Många satser

En viktig sak att komma ihåg är att när man skriver många satser efter varandra måste man separera dessa med ett komma. Detta gäller endast inuti funktioner, if-satser och loopar:

```
fun test
        if(1 == 1)
                a = 10,          #separerar varje sats med ett komma
                b = 5,
                print a+b
        endif,          #här måste ett komma finnas med för att separera if-satsen och loopen

        loop(i = 0 upto 5)
                c = 1+2*3,
                print c
        endloop
endfun
```

2.5 Exempelprogram

Nu när de flesta funktionerna i Nibla är förklarade är det dags att sätta ihop allt till ett fungerande program. Detta programmet kommer ta in en siffra och returnera om det är ett primtal eller inte:

```
fun isPrim(number)
        loop(i = 2 upto number – 1)
                if(number % i == 0)
                        return ”Not a prim”,
                        break
                endif
        endloop,

        if(number < 2)
                return 0        #bug i programmet gör att man inte kan returnera en sträng
        endif,

        return 1                #samma bug här men siffror går bra att returnera
endfun

#om man nu anropar denna funktionen och skriver ut det värdet som returneras:
print call isPrim(5)            # 1 kommer skrivas ut vilket betyder att 5 är ett primtal
print call isPrim(8)            # ”Not a prim” skrivs vilket betyder att 8 inte är ett primtal
```

3 Systemdokumentation

Systemdokumentationen kommer beskriva hur Nibla är uppbyggt och hur jag tänkte när jag implementerade de olika funktionerna.

3.1 Grammatik

Denna grammatiken är beskriven i BNF och beskriver hur Nibla är uppbyggt i sina delar

<PROGRAM> : <STMT_LIST>

<STMT_LIST> : <STMT_LIST> ”**,**” <STMT>
| <STMT>

<STMT> : <FUNCTION_DEF>
| <FUNCTION_CALL>
| <CONDITION>
| <REPETITION>
| <BREAK>
| <RETURN>
| <ASSIGN>
| <OUTPUT>
| <INPUT>
| <EXPR>

<FUNCTION_DEF> : ”**fun**” <IDENTIFIER> <STMT_LIST> ”**endfun**”
| ”**fun**” <IDENTIFIER> ”**(**” <PARA_LIST> ”**)**” <STMT_LIST> ”**endfun**”

<PARA_LIST> : String ”**,**” <PARA_LIST>
| String

<FUNCTION_CALL> : ”**call**” <IDENTIFIER> ”**(**” <CALL_LIST> ”**)**”
| ”**call**” <IDENTIFIER>

<CALL_LIST> : <EXPR> ”**,**” <CALL_LIST>
| <EXPR>

<CONDITION> : ”**if**” ”**(**” <COMP_EXPR> ”**)**” <STMT_LIST> <ELSE_CON>

<ELSE_CON> : ”**elseif**” ”**(**” <COMP_EXPR> ”**)**” <STMT_LIST> <ELSE_CON>
| ”**else**” <STMT_LIST> ”**endif**”
| ”**endif**”

<REPETITION> : ”**loop**” ”**(**” <RANGE_EXPR> ”**)**” <STMT_LIST> ”**endloop**”
| ”**loop**” ”**(**” <COMP_EXPR> ”**)**” <STMT_LIST> ”**endloop**”
| ”**loop**” ”**(**” <IDENTIFIER> ”**in**” <EXPR> ”**)**” <STMT_LIST>
”**endloop**”

<RANGE_EXPR> : <ASSIGN> ”**upto**” <EXPR>
| <ASSIGN> ”**downto**” <EXPR>

<COMP_EXPR> : <EXPR> <REAL_OPER> <EXPR>

<BREAK> : ”**break**”

<RETURN> : ”**return**” <EXPR>
| ”**return**”

<ASSIGN> : <IDENTIFIER> ”**=**” <EXPR>
| <IDENTIFIER> ”**add**” <EXPR>

<OUTPUT> : ”**print**” ”**(**” <MULT_OUTPUT> ”**)**”
| ”**print**” <EXPR>

<MULT_OUTPUT> : <EXPR> ”**,**” <MULT_OUTPUT>
| <EXPR>

<INPUT> : ”**input**” <IDENTIFIER>

<EXPR> : <EXPR> <ADD_OPER> <TERM>
| <TERM>

<TERM> : <TERM> <MULTI_OPER> <MODULU>
| <MODULU>

<MODULU> : <MODULU> <MOD_OPER> <FACTOR>
| <FACTOR>

<FACTOR> : <DIGIT>
| <STRING>
| <FUNCTION_CALL>
| <CONTAINER>
| <IDENTIFIER>
| ”**(**” <EXPR> ”**)**”

<ADD_OPER> : ”**+**”
| ”**-**”

<MULTI_OPER> : ”***”
| ”**/**”

<MOD_OPER> : ”**%**”

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

```
<STRING>        : /”[\w\W]*”/

<CONTAINER>     : <IDENTIFIER> ”[” <DIGIT> ”]”
                | ”[” <LIST> ”[” <DIGIT> ”]”
                | ”[” <LIST>
                | ”[” ”]”
                | ”{” ”}”
                | <IDENTIFIER ”[” String ”]” ”=” <EXPR>
                | <IDENTIFIER ”[” String ”]”

<LIST>          : <EXPR> ”,” <LIST>
                | <EXPR ”]”

<IDENTIFIER>    : <LETTER>
                | <LETTER> (<LETTER> | <DIGIT>)
                | <LETTER> <IDENTIFIER>

<LETTER>        : [a-zA-Z_]+

<DIGIT>         : Integer
```

3.2 Tokens

Tokens är en uppdelning av programtexten i delar. Dessa delar beskrivs med hjälp av reguljära uttryck. Nibla har några olika tokens som den använder för att matcha det som skrivs in, nämligen:

- *Whitespace* – Detta ignoreras vilket leder till att det är valfritt att använda eller ej
- *Integer* – Matchar heltalssiffror
- *Comments* – Kommentarer ignoreras eftersom det inte ska vara en del utav koden
- *Chars* – Matchar ord eller bokstäver från a till z. Både små och stora bokstäver
- *Strings* – Matchar strängar innanför två citationstecken t.ex *”string”*
- *Relations op* – Här matchas några relations operatorer som t.ex == (lika med)
- Everything else – Matchar varje tecken för sig precis som det är

3.3 Parsning

När ett datorspråk parsas utgår man ifrån många olika regler och beroende på vad som parsas kommer olika regler att spela roll och på så sätt kommer olika funktioner att inträffa. Dessa funktioner bildar tillsammans ett abstrakt syntaxträd med noder som sedan evalueras. Det är ingen skillnad när det kommer till Nibla när det gäller parsning. Jag var noga och tänkte mycket på grammatiken innan jag började implementera språket för att det skulle bli lättare att kunna skriva de regler som parsningen behövde. Detta hjälpte också eftersom när jag väl skulle skriva de regler som behövdes hade jag egentligen inte så stora problem med hur det skulle implementeras.

3.4 Klasser

För att kunna hålla reda på allt som händer i Nilba använder sig datorspråket av olika klasser som bestämmer vad som ska göras i olika situationer. Dessa klasser är skrivna i Ruby och hanterar det som skrivs i Nibla. När man skriver in ett kommando i Nibla skapas ett objekt utav en klass som håller reda på vad just detta kommando ska göra. När hela kommandot har parsats har ett abstrakt syntaxträd skapats som alla innehåller en funktion som kallas eval. Denna funktion anropas för varje objekt som har skapats.

3.5 Omgivning (scope)

Den omgivning, eller scope som det också kallas, som används i Nibla är mycket lik många andra datorspråk, dock är den inte implementerad på riktigt samma sätt. Nibla använder sig utav Dynamisk scope som fungerar till stor del. Detta går ut på att ha en huvud lista som i sin tur lagrar andra områden. På detta sätt kan man ”plocka ut” de variabler, eller liknande, som är giltiga i just det området man befinner sig. För varje ny if-sats, loop eller funktion man skapar kommer ett nytt område skapas som innehåller de områdena som ligger ”över” det nya området tillsammans med det område som nyss skapades. När man sedan lämnar t.ex en funktion kan man inte längre använda sig utav det området som skapades i funktionen. Med andra ord kan man bara använda de variabler som skapades ”utanför” och i det område man befinner sig i.
Här följer ett exempel på hur omgivningen ser ut vid utskriften inuti en if-sats:

```
a = 5
b = 10
if(1==1)
        c = 15,
        print (a, b, c)          {3=>{2=>{a=>5, b=>10}, c=>15}}
endif
```

3.6 Vad som inte fungerar som tänkt

Även om allt fungerar om man gör på ett visst sätt finns det fortfarande saker som inte fungerar precis som jag hade tänkt mig. Det är främst småsaker som inte fungerar som t.ex. funkar det att returnera strängar ibland från funktioner och ibland funkar det inte. Inte heller kommentarer fungerar.

Ett lite större problem som inte spelar så stor roll i de flesta fallen handlar om omgivningen, eller scopeet, som är implementerad på ett sätt som gör att det inte fungerar i riktigt alla lägen. Här följer ett exempel på när den inte fungerar:

```
        fun a(x)
                fun b(y)
                        print y+x
                endfun

                fun c(x)
                        call b(5)
                endfun

                call c(10)
        endfun

        call a(2)
```

När koden *call a(2)* körs anropas funktion a och variabeln x får värdet 2. Sedan körs koden *call c(10)* som anropar funktionen c och en ny variabel x får värdet 10. Efter det körs koden *call b(5)* som anropar funktion b och variabeln y får värdet 5. Nu körs koden *print y+x* som ska skriva ut y+x. Det jag vill att den ska skriva ut är 5+2 som är 7. Men istället för det x:et som skapades för

funktion a så försöker den ta x:et från funktion c. Eftersom scopet är dynamiskt och inte statiskt kommer man få ett värde på x som man kanske inte förväntar sig.

4 Utvärdering

Jag kommer diskutera vad jag har lärt mig, hur jag tyckte detta arbetet gått och varför jag gjorde som jag gjorde vid vissa tillfällen.

4.1 Tankar kring kursen vid start

Redan under slutet av höstterminen började jag fundera lite på vad det skulle innebära att skapa ett eget datorspråk. Detta var många tankar som flög i huvudet men dem främsta var om att jag trodde det skulle bli mycket svårt, nästan som en omöjlighet. Sedan när kursen TDP007 började på vårterminen fick jag lära mig datorspråket Ruby som kom till stor hjälp när det väl var dags att implementera sitt egna datorspråk. Jag fick också lära mig hur man använder sig utav RD-parsern vilket jag har använt för att just parsa Nibla. Jag hade också föreläsningar om hur man skulle gå till väga för att skapa ett eget språk och sakta men säkert kom det upp idéer och planer om hur jag ville att mitt språk skulle se ut. Allt detta gjorde också att mina första tankar om att det nästan skulle bli omöjligt att skapa sitt eget språk var som bortblåsta. Självklart visste jag att det skulle bli lite klurigt men samtidigt visste jag att det skulle bli långt ifrån omöjligt vilket det inte heller blev.

4.2 Tankar under kursens gång

När kursen hade hållit på ett tag började jag inse mer och mer hur jag ville att mitt egna språk skulle se ut. Jag hade tänkt mycket på hur mitt språk skulle se ut och även om grammatiken var bra formulerad tidigt var det inte förens någonstans i mitten av kursen då jag visste precis hur jag ville att allting skulle se ut. Även om det var spännande och kul i början var det inte förrän nu som jag verkligen tyckte det var väldigt intressant eftersom jag nu hade all information som jag behövde för att kunna skapa det jag ville skapa. Dock stötte jag på problem och fastnade på en del saker som tog tid att lösa men det vore ju konstigt om jag inte hade stött på några problem.

4.3 Tankar i slutet av kursen

När jag hade kommit så långt så jag hade ett ”färdigt” datorspråk var jag ganska nöjd. Även om jag hade velat implementera vissa saker bättre och fixa saker som inte riktigt fungerade tyckte jag ändå att jag hade gjort ett bra jobb. Jag hade ju fått med allt det som jag ville få med och faktiskt lite till.

4.4 Problem under kursen

Under projektets gång hår jag stött på några olika problem som jag har fått stanna upp och fundera lite extra över. Det första problemet jag stötte på kom tidigt eftersom det handlade om vad för något slags språk jag ville skapa. Detta problemet var inte så stort eftersom jag bestämde mig tillslut efter att ha funderat på några olika alternativ att jag ville göra ett ”vanligt” datorspråk. Anledningen till att jag valde att göra ett språk som är ganska lika andra språk var just för att lära mig om hur andra datorspråk fungerar. Denna kursen var ju inte till för att skapa ett eget språk som ska bli lika stort som t.ex C++ utan det var först och främst till för att få en förståelse för hur andra datorspråk fungerar vilket jag också tycker att jag fick.

Efter detta handlade mina största problem om implementationsstadiet eftersom det var en av de stora delarna av kursen. Bland annat fick jag problem när jag skulle implementera looparna. Jag satt och tänkte ganska många dagar på hur jag skulle kunna skapa en automatisk uppräkning av den variabel man deklarerar i den ena loopen. Detta var en sådan enkel lösning tillslut att jag faktiskt tyckte jag var ganska trög. Jag hade ju en variabelklass som satte variabler. Allt jag behövde göra var ju att anropa den med samma namn på variabeln och öka värdet med 1.

Jag hade också problem med funktionerna och hur dessa skulle kunna anropas med endast rätt antal parametrar. Men efter några timmars knep och knop lyckades jag komma på hur jag skulle lösa det. Nämligen genom att spara antalet parametrar i en lista när funktionen deklarerades och sedan använda denna listan för att deklarera variabler för de parametrar man skickade in. På så sätt kunde jag också jämföra antal parametrar som funktionen hade och hur många parametrar man skickade med.

Förutom detta som jag har tagit upp har jag fastnat på lite mindre problem som löstes ganska snabbt. Bland annat hur strängar skulle implementeras eller hur de mattematiska prioriteringarna skulle se ut för att följa standarden som finns idag. Dock har det ändå flutit på ganska bra och i slutet fick jag ett färdigt resultat.

4.5 Vad jag har lärt mig

I hela den här kursen har jag lärt mig hur man skapar ett eget datorspråk. Eftersom kursen har gått igenom de flesta steg för att skapa ett datorspråk har jag fått stor förståelse för hur många andra språk fungerar i grunden. Jag har också blivit en liten bättre programmerare eftersom jag har lärt mig nya knep för att göra olika saker som . Detta projektet har också lärt mig att just arbeta med projekt. Planering är viktigt och ju noggrannare man är i början desto mindre tid behöver man spendera på att gå tillbaka och ändra eller lägga till.

5 Bilagor

I dessa bilagor finns den fil som har hand om parsningen av den kod som man skriver in i Nibla. Den fil som innehåller alla klasser finns också med här.

5.1 Nibla_Parser.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

require './rdparse.rb'
require './Nibla_Code.rb'

class Nibla

  def initialize
    @nibla = Parser.new("Nibla") do
      token(/\s+/)              #Ignore whitespace
      token(/\d+/) {|x| x.to_i } #Integer
      token(/\#.*/)             #Comment
      token(/[a-zA-Z]+/) {|x| x.to_s} #Chars
      token(/"[\w\W]*"/) {|x| x.to_s} #Strings
      token(/\=\=/) {|x| x}      #match relation op
      token(/\!\=/) {|x| x}
      token(/\>\=/) {|x| x}
      token(/\<\=/) {|x| x}
      token(/./) {|x| x }        #Let everything else be as it is

      start :program do
        match(:stmt_list) {|stmts| Final_eval.new(stmts)}
      end

      rule :stmt_list do
        match(:stmt_list, ',', :stmt) {|a, _, c| MultiStmt.new([a,c])}
        match(:stmt) {|a| a}
      end

      rule :stmt do
        match(:function_def)
        match(:function_call)
        match(:condition)
        match(:repetition)
        match(:break)
        match(:return)
        match(:assign)
        match(:output)
        match(:input)
        match(:expr)
      end
```

```
    rule :function_def do
      match('fun', :identifier, :stmt_list, 'endfun') {|_, b, c, _| SetVar.new(b, FunctionDef.new(c))}
      match('fun', :identifier, '(', :para_list, ')', :stmt_list, 'endfun') {|_, b, _, d, _, f, _| SetVar.new(b,
FunctionDef.new(f, d))}
    end

    rule :para_list do
      match(String, ',', :para_list) {|a, _, c| [a, c].flatten}
      match(String) {|a| [a]}
    end

    rule :function_call do
      match('call', :identifier, '(', :call_list, ')') {|_, b, _, d, _| FunctionCall.new(b, d)}
      match('call', :identifier) {|_, b| FunctionCall.new(b)}
    end

    rule :call_list do
      match(:expr, ',', :call_list) {|a, _, c| [a, c].flatten}
      match(:expr) {|a| [a]}
    end

    rule :condition do
      match('if', '(', :comp_expr, ')', :stmt_list, :else_con) {|_, _, c , _, e, f| If.new(c, e, f)}
    end

    rule :else_con do
      match('elseif', '(', :comp_expr, ')', :stmt_list, :else_con) {|_, _, c, _, e, f| If.new(c, e ,f)}
      match('else', :stmt_list, 'endif') {|_, b, _| b}
      match('endif')
    end

    rule :repetition do
      match('loop', '(', :range_expr, ')', :stmt_list, 'endloop') {|_, _, c, _, e, _| Loop1.new(c[0], c[1],
c[2], e)}
      match('loop', '(', :comp_expr, ')', :stmt_list, 'endloop') {|_, _, c, _, e, _| Loop2.new(c, e)}
      match('loop', '(', :identifier, 'in', :expr, ')', :stmt_list, 'endloop') {|_, _, c, _, e, _, g, _|
Loop3.new(c, e, g)}
    end

    rule :range_expr do
      match(:assign, 'upto', :expr) {|a, b, c| [a, b, c]}
      match(:assign, 'downto', :expr) {|a, b, c| [a, b, c]}
    end

    rule :comp_expr do
      match(:expr, :real_oper, :expr) {|a, b, c| Relation.new(a, b, c)}
    end

    rule :break do
      match('break') {Break.new}
```

```
end

rule :return do
  match('return', :expr) {|_, b| Return.new(b)}
  match('return') {Return.new}
end

rule :assign do
  match(:identifier, '=', :expr) {|a, _, c| SetVar.new(a, c)}
  match(:identifier, 'add', :expr) {|a, _, c| AddList.new(a, c)}
end

rule :output do
  match('print', '(', :mult_output, ')') {|_, _, c, _| Output.new(c)}
  match('print', :expr) {|_, b| Output.new([b])}
end

rule :mult_output do
  match(:expr, ',', :mult_output) {|a, _, c| [a, c].flatten}
  match(:expr) {|a| [a]}
end

rule :input do
  match('input', :identifier) {|_, a| Input.new(a, gets)}
end

rule :expr do
  match(:expr, :add_oper, :term) {|a, b, c|  Expression.new(a, b, c)}
  match(:term)
end

rule :term do
  match(:term, :multi_oper, :modulu) {|a, b ,c| Expression.new(a, b, c)}
  match(:modulu)
end

rule :modulu do
  match(:modulu, :mod_oper, :factor) {|a, b, c| Expression.new(a, b, c)}
  match(:factor)
end

rule :factor do
  match(:digit)
  match(:string)
  match(:function_call)
  match(:container)
  match(:identifier)
  match('(', :expr, ')') {|_,a,_| a}
end
```

```
rule :add_oper do
  match('+')
  match('-')
end

rule :multi_oper do
  match('*')
  match('/')
end

rule :mod_oper do
  match('%')
end

rule :real_oper do
  match('==')
  match('!=')
  match('>')
  match('<')
  match('>=')
  match('<=')
end

rule :string do
  match(/'[\w\W]*'/) {|a| String.new(a)}
end

rule :container do
  match(:identifier, '[', :digit, ']') {|a, _, c, _| List.new(a, c)}
  match('[', :list, '[', :digit, ']') {|_, b, _, d, _| List.new(b, d)}
  match('[', :list) {|_, b| List.new(b)}
  match('[', ']') {List.new}

  match('{', '}') {|_, b| Dict.new({})}
  match(:identifier, '[', String, ']', '=', :expr) {|a, _, c, _, _, f| Dict.new(a, c, f)}
  match(:identifier, '[', String, ']') {|a, _, c, _| Dict.new(a, c)}
end

rule :list do
  match(:expr, ',',  :list) {|a, _, c| [a, c].flatten}
  match(:expr, ']') {|a, _| [a]}
end

rule :identifier do
  match(:letter)
  match(:letter, (:letter or :digit))
  match(:letter, :identifier)
end

rule :letter do
```

```
      match(/^[a-zA-Z_]+/) {|a| Variable.new(a)}
    end

    rule :digit do
      match(Integer) {|a| Digit.new(a)}
    end
  end
end

def eval_file(str)
  file = File.open(str.split[1])
  parse = true
  counter = 0
  for line in file
    if line != "\n"
      if parse and counter == 0
        str = line
      else
        str += line
      end

      if line =~ /^if|^loop|^fun/
        parse = false
        counter += 1
      end

      if line =~ /^endif|^endloop|^endfun/
        parse = true
        counter -= 1
      end

      if parse and counter == 0
        begin
          @nibla.parse(str).eval
        rescue Exception => error
          puts error.message
          break
        end
      end
    end
  end
end

def exit?(str)
  ["bye"].include?(str.chomp)
end

def start
  print "Nibla <> "
  str = gets
```

```
    @nibla.logger.level = Logger::WARN
    if exit?(str)
      puts "Bye Bye"
    elsif str =~ /^open\s+.+\.nibla/
      eval_file(str)
      start
    else
      begin
        @nibla.parse(str).eval
      rescue Exception => error
        puts error.message
      end
      start
    end
  end
end

Nibla.new.start
```

5.2 Nibla_Code.rb

```
#!/usr/bin/env ruby
# -*- coding: utf-8 -*-

class Scope
  def initialize
    @@scope_counter = 1
    @@scope = Hash.new
  end

  def Scope.setScope(scope)
    @@scope = scope
  end

  def Scope.getScope
    @@scope
  end

  def Scope.getCounter
    @@scope_counter
  end

  def Scope.newScope
    scope = Hash.new
    @@scope_counter += 1
    scope[@@scope_counter] = Scope.getScope
    return scope
  end

  def Scope.setOldScope(scope)
    Scope.setScope(scope[@@scope_counter])
    @@scope_counter -= 1
  end
end

Scope.new

class Final_eval
  def initialize(stmts)
    @stmts = stmts
  end

  def eval
    if @stmts.respond_to? "each"
      @stmts.each {|stmt| stmt.eval}
    else
      @stmts.eval
    end
  end
```

```
end

class MultiStmt
  def initialize(stmts)
    @stmts = stmts
  end

  def eval
    for i in @stmts
      value = i.eval
      if value == 'break'
        return 'break'
      elsif value.class == Array
        if value[0] == 'return'
          return value
        end
      end
    end
  end
end

class FunctionDef
  def initialize(stmt, para = [])
    @para = para
    @stmt = stmt
    @scope = Hash.new
  end

  def eval
    value = @stmt.eval
    if value.class == Array
      if value[0] == 'return'
        return value
      end
    end
  end

  def getPara
    if @para == nil
      return []
    else
      return @para
    end
  end
end

class FunctionCall
  def initialize(name, para = [])
    @name = name
    @para = para
```

```
  end

  def eval
    @scope = Scope.newScope
    Scope.setScope(@scope)
    paras = @name.eval.getPara
    if paras.length == @para.length
      for i in 0..(@para.length - 1)
        if @para[i].eval.class == Fixnum
          SetVar.new(Variable.new(paras[i]), Digit.new(@para[i].eval)).eval
        else
          SetVar.new(Variable.new(paras[i]), String.new(@para[i].eval)).eval
        end
      end
      value = @name.eval.eval
      if value.class == Array
        if value[0] == 'return'
          Scope.setOldScope(@scope)
          return value[1]
        end
      end
    else
      raise "Expected #{paras.length} parameters but got #{@para.length}"
    end
    Scope.setOldScope(@scope)
  end
end

class If
  def initialize(rel_expr, stmt, else_stmt)
    @rel_expr = rel_expr
    @stmt = stmt
    @else_stmt = else_stmt
    @scope = Hash.new
  end

  def eval
    @scope = Scope.newScope
    Scope.setScope(@scope)
    if @rel_expr.eval
      value = @stmt.eval
      if value == 'break'
        Scope.setOldScope(@scope)
        return 'break'
      elsif value.class == Array
        if value[0] == 'return'
          Scope.setOldScope(@scope)
          return value
        end
      end
```

```
    else
      value = @else_stmt.eval
      if value == 'break'
        Scope.setOldScope(@scope)
        return 'break'
      elsif value.class == Array
        if value[0] == 'return'
          Scope.setOldScope(@scope)
          return value
        end
      end
    end
    Scope.setOldScope(@scope)
  end
end

class Loop1
  def initialize(var, direction, range, stmt)
    @var = var
    @direction = direction
    @range = range
    @stmt = stmt
    @scope = Hash.new
  end

  def eval
    @scope = Scope.newScope
    Scope.setScope(@scope)

    name = @var.name.var
    value = @var.eval

    if @direction == 'upto'
      value.upto(@range.eval) do
        stmt_value = @stmt.eval
        if stmt_value == 'break'
          break
        elsif stmt_value.class == Array
          if stmt_value[0] == 'return'
            Scope.setOldScope(@scope)
            return stmt_value
          end
        end
        value += 1
        SetVar.new(Variable.new(name), Digit.new(value)).eval
      end
    else
      value.downto(@range.eval) do
        stmt_value = @stmt.eval
        if stmt_value == 'break'
```

```
        break
      elsif stmt_value.class == Array
        if stmt_value[0] == 'return'
          Scope.setOldScope(@scope)
          return stmt_value
        end
      end
      value -= 1
      SetVar.new(Variable.new(name), Digit.new(value)).eval
    end
  end

  Scope.setOldScope(@scope)
 end
end

class Loop2
 def initialize(comp_expr, stmt)
  @comp_expr = comp_expr
  @stmt = stmt
  @scope = Hash.new
 end

 def eval
  @scope = Scope.newScope
  Scope.setScope(@scope)

  while @comp_expr.eval
   value = @stmt.eval
   if value == 'break'
    break
   elsif value.class == Array
    if value[0] == 'return'
     Scope.setOldScope(@scope)
     return value
    end
   end
  end

  Scope.setOldScope(@scope)
 end
end

class Loop3
 def initialize(iter_var, iter_stmt, stmt)
  @iter_var = iter_var
  @iter_stmt = iter_stmt
  @stmt = stmt
  @scope = Hash.new
 end
```

```
  def eval
    @scope = Scope.newScope
    Scope.setScope(@scope)

    if @iter_stmt.eval.class == Fixnum
      raise "Error: Can not iterate through numbers"
    else
      name = @iter_var.var

      for i in 0..(@iter_stmt.eval.length - 1)
        SetVar.new(Variable.new(name), String.new(@iter_stmt.eval[i])).eval
        value = @stmt.eval
        if value == 'break'
          break
        elsif value.class == Array
          if value[0] == 'return'
            Scope.setOldScope(@scope)
            return value
          end
        end
      end
    end

    Scope.setOldScope(@scope)
  end
end

class Break
  def eval
    'break'
  end
end

class Return
  def initialize(expr = nil)
    @expr = expr
  end

  def eval
    ['return', @expr.eval]
  end
end

class SetVar
  attr_reader :name
  def initialize(name, expr)
    @name = name
    @expr = expr
    @scope = Hash.new
```

```
  end

  def eval
    @scope = Scope.getScope
    @namedVar = @name.var
    value = @expr
    if @expr.class != FunctionDef
      value = @expr.eval
    end

    counter = Scope.getCounter
    while counter > 1
      if @scope[counter].has_key? @namedVar
        @scope[counter][@namedVar] = value
        return
      else
        @scope = @scope.values[0]
        @scope = Hash[*@scope.collect {|x| [x]}.flatten]
        counter -= 1
      end
    end

    @scope = Scope.getScope
    @scope[@namedVar] = value
  end
end

class AddList
  def initialize(name, expr)
    @name = name
    @expr = expr
  end

  def eval
    @scope = Scope.getScope
    @namedVar = @name.var

    if @name.eval.class != Array
      raise "Can only add values in lists"
    end

    value = @expr
    if @expr.class != FunctionDef
      value = @expr.eval
    end

    counter = Scope.getCounter

    if counter == 1
      if @scope.has_key? @namedVar
```

```
        @scope[@namedVar] = @name.get_value << value
        return
      end
    else
      while counter > 1
        if @scope[counter].has_key? @namedVar
          @scope[counter][@namedVar] = @name.get_value << value
          return
        else
          @scope = @scope.values[0]
          @scope = Hash[*@scope.collect {|x| [x]}.flatten]
          counter -= 1
        end
      end
    end

    raise "#{@namedVar} can not add #{value}"
  end
end

class Output
  def initialize(value)
    @value = value
  end

  def eval
    if @value.class == Array
      for i in @value
        puts check_var(i)
      end
    else
      puts check_var(@value)
    end
  end
end

class Input
  def initialize(name, value)
    @name = name
    @value = value
  end

  def eval
    if @value =~ /[0-9]+/
      SetVar.new(@name, Digit.new(instance_eval(@value))).eval
    else
      SetVar.new(@name, String.new(instance_eval(@value))).eval
    end
  end
end
```

```
class Relation
  def initialize(left_expr, op, right_expr)
    @left_expr = left_expr
    @right_expr = right_expr
    @op = op
  end

  def eval
    case @op
    when '==' then return (@left_expr.eval == @right_expr.eval)
    when '!=' then return (@left_expr.eval != @right_expr.eval)
    when '>' then return (@left_expr.eval > @right_expr.eval)
    when '<' then return (@left_expr.eval < @right_expr.eval)
    when '>=' then return (@left_expr.eval >= @right_expr.eval)
    when '<=' then return (@left_expr.eval <= @right_expr.eval)
    end
  end
end

class Expression
  def initialize(var1, op, var2)
    @var1 = var1
    @var2 = var2
    @op = op
  end

  def eval
    case @op
    when '*' then return (@var1.eval * @var2.eval)
    when '/' then return (@var1.eval / @var2.eval)
    when '+' then return (@var1.eval + @var2.eval)
    when '-' then return (@var1.eval - @var2.eval)
    when '%' then return (@var1.eval % @var2.eval)
    end
  end
end

class Digit
  def initialize(number)
    @value = number
  end

  def eval
    return @value
  end
end

class String
  def initialize(str)
```

```
    @value = str
  end

  def eval
    return @value
  end
end

class List
  def initialize(list = nil, pos = nil)
    @list = list
    @pos = pos
  end

  def eval
    if @list == nil
      return []
    end

    return_list = []
    if @pos == nil
      for i in @list
        return_list << i.eval
      end
      return return_list
    else
      if @list.class == Array
        return @list[@pos.eval].eval
      else
        return @list.eval[@pos.eval]
      end
    end
  end
end

class Dict
  def initialize(dict, key = nil, value = nil)
    @dict = dict
    @key = key
    @value = value
  end

  def eval
    if @key != nil and @value != nil
      @dict.eval[@key] = @value.eval
    elsif @key != nil
      return @dict.eval[@key]
    else
      return @dict
    end
```

```
  end
end

class Variable
  attr_reader :var
  def initialize(var)
    @var = var
  end

  def eval
    check_var(self)
  end

  def get_value
    return check_var(self)
  end
end

def check_var(var)
  scope = Scope.getScope
  if notvar(var)
    return var.eval
  end

  if var.class == Variable
    varName = var.var
  else
    varName = var
  end

  if scope.has_key? varName
    return scope[varName]
  elsif scope[Scope.getCounter] != nil
    counter = Scope.getCounter
    while counter != 1
      if scope[counter].has_key? varName
        return scope[counter][varName]
      else
        scope = scope.values[0]
        scope = Hash[*scope.collect {|x| [x]}.flatten]
        counter -= 1
      end
    end
  end

  raise "Error: The variable #{varName} has no value"
end

def notvar(var)
  classes = [Digit, String, Expression, List, Dict, FunctionCall]
```

```
  if classes.include?(var.class)
    return true
  end
  return false
end
```