RIPPLE Interpreted Prefix Programming Language

- En introduktion

TDP019 - Vårterminen 2011

Stefan Nyman
steny228@student.liu.se

Sammanfattning

Programmeringsspråket RIPPLE är ett relativt konventionellt språk med de grundläggande egenskaper man kan förvänta sig, men besitter dock den egenskapen att matematiska och logiska uttryck skrivs i polsk eller sk. prefixnotation. Därav kommer språkets namn *RIPPLE Interpreted Prefix Programming Language*. Av namnet kan två viktiga egenskaper utläsas, dels är språket interpreterat, dels används prefixnotation som tidigare nämnts. Andra viktiga egenskaper hos språket är att det är imperativt, till viss del hårt typat och att det bygger på att man i alla lägen använder sig av funktioner för att strukturera källkoden.

Innehåll

1 Inledning

Denna rapport har som mål att för läsaren presentera programmeringsspråket RIPPLE, vilket har designats och implementerats som en del i kursen *TDP019 Projekt: Datorspråk* på Linköpings universitet. Arbetet utfördes under våren 2011, vilket var programmet Innovativ Programmerings andra termin. Rapporten kommer att behandla områden som ren information om språket, dess utvecklingsprocess, koncept, praktiska användande och egna funderingar om programmeringsspråk i allmänhet.

1.1 Syfte

Rapportens syfte kan delas upp i två distinkta delar. Det primära syftet är att ge läsaren en stabil grund att stå på både när det gäller teoretisk kunskap om och praktiskt användande av språket RIPPLE. Ett mer sekundärt syfte består i att dokumentera språket med dess ingående beståndsdelar, samt att ge en insyn i författarens tankar om utvecklingsprocesser, programmeringsspråk och programmering.

1.2 Målgrupp

Avsedda läsare av denna rapport är medstuderande på programmet för Innovativ Programmering, lärare samt personer med intresse för programmeringsspråk. Då målgruppen är ganska snäv kommer det i rapporten att antagas att läsaren förstår, eller i alla fall har en uppfattning om, de koncept som diskuteras och de tankegångar som förs.

1.3 Disposition

Rapporten är uppdelad i fem distinkta delar:

Inledning:	I denna del läggs grunden för resten av rapporten.
Användarhandledning:	Här presenteras hur man som användare kan ta del av projektet och själv börja skapa program skrivna i RIPPLE
Systemdokumentation:	Avsnitt presenterade i den här delen ger en mer teknisk bild över hur RIPPLE fungerar, dvs. hur det går till från att man skrivit sin kod tills det att programmet har körts.
Programkod:	Den kod som driver RIPPLE presenteras här.
Erfarenheter & reflektion:	Här tar jag upp mina egna funderingar kring programmeringsspråk och utvecklingsprocessen.

2 Användarhandledning

Denna del av rapporten är tänkt att ge läsaren en grund för att utveckla program med RIPPLE. Handledningen kommer att börja ganska grundläggande med en del där språket beskrivs i fråga om vilka tankar som genomsyrat projektet under utvecklingsperioden och varför språket ser ut som det gör. Vidare förklaras hur man på ett smidigt sätt kan få till *syntaxmarkering*[1] och hur man faktiskt programmerar något. I denna handledning förutsätts det att du använder GNU/Linux i någon form och att Ruby är av version 1.9.2 eller nyare.

2.1 Om RIPPLE

Som namnet antyder är RIPPLE ett interpreterat programmeringsspråk där viss del av syntaxen uttrycks i prefixnotation. Denna notation är inte så vanlig men finns i språk som till exempel *Lisp* och liknande språk. Fördelen med att använda sig av denna notation är att ordningen för operatorerna blir entydig, och det framgår på så sätt vad som skall ske och i vilken ordning.

Den kanske viktigaste egenskapen hos RIPPLE, till skillnad från många andra interpreterade språk, är att all kod som skrivs måste skrivas inom funktionsdeklarationer. Anledningen till detta designval är helt enkelt att all kod bör ha någon logisk anknytning till sin omgivning. Genom att tvinga användaren att skriva sin kod i funktioner blir koden renare och det finns ingen tvetydighet gällande var och när en viss variabel ändras.

2.1.1 Programstruktur

Ett normalt program i RIPPLE har några regler som måste efterföljas. Dessa är i korthet:

- All kod måste vara skriven i funktioner.
- Det måste finnas en funktion med namnet *init*.

Exempel på felaktig programstruktur:

```
x int: 4;
defun init ->
  println x;
;
```

[1] Försvenskning av det engelska uttrycket *Syntax Highlighting*

Anledningen till att denna struktur är felaktig är att det i det här fallet deklareras en variabel x utanför en funktion. För att denna kod skall kunna köras måste deklarationen av variabeln flyttas in i funktionskroppen *init*:

```
defun init ->
  x int: 4;
  println x;
;
```

2.2 Syntaxmarkering

I katalogen för RIPPLE finns en fil med namn *ripple-mode.el* som ger användaren möjlighet att aktivera *syntaxmarkering* i Emacs. När detta *mode*[2] används ges även viss hjälp med indentering av kod genom att man använder Tab-tangenten. Det är dock viktigt att observera att detta bara fungerar i editorn Emacs och inget annat. Har du inte Emacs rekommenderar jag att du installerar den, om inte annat bara för att skriva fin kod i RIPPLE.

2.2.1 Installation av ripple-mode

För att på ett enkelt sätt kunna använda sig av *ripple-mode.el* bör du kopiera den till din *.emacs.d*-katalog. Exempelvis:

```
~/.emacs.d/els/
```

Om mappen inte finns skapas den enkelt med kommandot:

```
mkdir -p ~/.emacs.d/els
```

Kopiera sedan *ripple-mode.el* till denna katalog:

```
cp ripple-mode.el ~/.emacs.d/els/
```

Nu är det dags att konfigurera Emacs så att man kan använda ripple-mode när man skriver källkod. För att göra detta öppnar du filen *.emacs* i din hemkatalog och skriver in följande rader:

```
(add-to-list 'load-path "~/.emacs.d/els/")
(require 'ripple-mode')
```

Genom att nu döpa dina filer med filändelsen ".rip" kommer dessa nu att visas med *syntaxmarkering* i Emacs.

[2]Kan i det här fallet likställas med en inställning i editorn

2.3 Programmering med RIPPLE

Programmering i RIPPLE kan utföras på två olika sätt. Antingen väljer man att använda sig av den interaktiva tolken eller så programmerar man på ett mer traditionellt sätt genom att skriva in kod i en textfil och sedan köra den. Nedan ges instruktioner för hur man går tillväga oavsett vilken metod man väljer.

2.3.1 Med interaktiv tolk

Den interaktiva tolken är i sig ett program som sparar ner den kod du skriver in och sedan vid givet kommando försöker exekvera densamma. I korthet kan man se tolken som en buffert där all kod samlas i väntan på körning.

För att starta tolken anger du kommandot:

```
ruby ripple.rb
```

Genom att tolken nu är startad går det bra att direkt börja skriva in kod. Om man skulle behöva hjälp under arbetet kan man skriva in kommandot *help* och på så sätt få tillgång till en samling med kommandon för editering av bufferten.

2.3.2 Utan tolk

Det kanske vanligaste sättet att programmera på är ju som bekant att man skriver sin kod i en textfil för att sedan köra den genom något verktyg, antingen för att kompilera koden eller för att interpretera den. RIPPLE utnyttjar även denna teknik. Fördelen med denna form av kodande är att man slipper modifiera bufferten med kommandon och man kan på så sätt snabbare utveckla sin kod. För att köra sin kod på detta sätt anropar man RIPPLE på följande sätt:

```
ruby ripple.rb <filnamn>
```

Där <filnamn> ersätts med den aktuella källkodsfilen.

2.3.3 Datatyper

RIPPLE har ett relativt begränsat urval datatyper och de bör inte vara obekanta även för en relativt oerfaren programmerare. De datatyper som finns specificerade är: *int*, *float*, *bool* och *string*. Vidare finns även typen *void* för att kunna deklarera att funktioner inte skall returnera något.

2.3.4 Variabler och tilldelning

Variabler i RIPPLE består av tecknen a-z, A-Z samt tecknet "_" och kan vara av godtycklig längd, dock minst en bokstav. När man skall tilldela dessa variabler ett värde, eller bara deklarera dem, skiljer sig notationen i RIPPLE från de flesta programmeringsspråk. I ett "vanligt" språk kan det se ut som följer:

```
int x = 3;
```

Detta är inte fallet i RIPPLE då tecknet "=" inte används för tilldelning utan istället för att göra jämförelser. Motsvarande kod i RIPPLE skulle alltså bli:

```
x int: 3;
```

Skillnaden här är alltså att tecknet ":" används för tilldelning och att typen på variabeln anges efter variabelnamnet. Skulle man bara vilja deklarera en variabel kan man hoppa över tilldelningsbiten av uttrycket, men då är det viktigt att komma ihåg att man inte tilldelat variabeln något värde innan man försöker använda den.

```
x int;
```

2.3.5 Operatorer och operationer

För att kunna göra aritmetiska och logiska operationer behöver ett programmeringsspråk ett antal operatorer som betyder olika saker. RIPPLE delar upp dessa operatorer i aritmetik-, logik- och relationsoperatorer.

- Aritmetiska operatorer: $+$, $-$, $*$, $/$, $\%$ och $\hat{}$.
- Logiska operatorer: $=$, !, $|$ och &.
- Relationsoperatorer: $<$, $>$, $<=$ och $>=$.

Med undantag för "!" tar alla operatorer två operander och man kan på så sätt sätta ihop lite olika operationer som följande:

```
Aritmetik:
(+ 1 2)
(+ 1 (* 3 (/ 3 1)))
(^ 2 4)
```

```
Logik:
(& true false)
(| true false)
(! (| false (& true (! true))))

Relationer:
(> 3 1)
(<= 2 3)
```

Då relationsoperationerna resulterar i sant eller falskt kan dessa även kombineras med de vanliga logiska operationerna som till exempel:

```
(! (> 3 2))
(& (>= 3 3) (| (! false) (> 3 4)))
```

2.3.6 In- och utmating

För in- och utmatning erbjuder RIPPLE funktionalitet för att både läsa information från tangentbord och filer, samt att skriva ut information till terminalen och till fil. För att läsa från tangentbord och fil används samma kommando, vilket kallas för *read*.

Exempel på hur man läser in från tangentbordet och sparar i en variabel:

```
str string;
read(stdin, str);
```

Notera att ”stdin” angavs som källa och det betyder alltså att man läser från standardenheten, eller med andra ord tangentbordet. För att läsa från fil anger man istället filnamnet som källa på följande sätt:

```
str string;
read("filnamn.txt", str);
```

För att skriva ut information till terminalen används kommandona *print* eller *println*. Skillnaden mellan dessa är som ändelsen antyder att *println* gör radbrytning. För att skriva till fil används kommandot *printf* enligt följande:

```
str string: "en textsträng";
printf(str, "målfil.txt");
```

För att bara skriva ut till terminalen kan man göra:

```
str string: "en textsträng";
print str; #detta kommer inte ge en ny rad
println str; #detta ger ny rad
```

2.3.7 Funktioner

Som tidigare nämnts bygger program skrivna i RIPPLE på funktioner. Med andra ord kan man säga att det är funktioner som bygger upp programmet och det är funktionernas anrop till varandra som gör att något händer. Funktioner kan deklareras antingen med specificerad returtyp eller utan, med parameterlista eller utan, samt en kombination av de båda. Det viktigaste när man deklarerar en funktion är att om målet med den är att returnera ett värde måste funktionen ha en specificerad returtyp. Funktioner i sig består av ett godtyckligt antal så kallade *statements*, det är dessa som ser till att något händer i programmet.

Exempel på giltiga funktionsdeklarationer:

```
defun int add_two(int a) ->
  return (+ a 2);
;

defun print_something(string str) ->
  println str;
;

defun hello ->
  print "Hello, World!";
;

defun int get_one ->
  return 1;
;
```

Något som även är värt att notera är att det är tillåtet att deklarera funktioner i funktioner. Detta gör att funktionerna blir lokala för den funktion de ligger inom. Med andra ord kan inte en funktion komma åt en funktion inuti en annan funktion, men en funktion inuti en funktion kan komma åt funktioner utanför den funktion den befinner sig inom.

Exempel på funktioner i funktioner:

```
defun void init ->
  defun int add_two(int a) ->
    return (+ a 2);
  ;;
  defun void print_number ->
    println add_two(3);
```

```
  ;;
  print_number();
;
```

I ovanstående exempel deklareras två funktioner i funktionen *init*. Dessa funktioner kan fritt använda sig av varandra vilket i just det här exemplet demonstreras genom att funktionen *print_number* skriver ut returvärdet från funktionen *add_two*. Just detta exempel är ett lätt sådant och det finns inget som hindrar en från att definiera fler funktioner i dessa funktioner.

Något som även är värt att notera i exemplet ovan är att funktionsdeklarationerna i huvudfunktionen avslutas med två ”;” istället för ett. Detta beror sig av att funktionsdeklarationer när de skrivs i andra funktioner hanteras som statements och skall således avslutas med ”;”. På så sätt blir koden tydligare och man kan enkelt se funktionsdeklarationerna.

2.3.8 For-loopar

For-loopar i RIPPLE uppför sig ungefär på samma sätt som i andra språk, det vill säga att man anger ett startvärde, ett slutvärde och ett inkrementeringsvärde. I RIPPLE motsvaras dessa av en variabel, ett logiskt uttryck och ett valbart inkrementeringsvärde.

Exempel på godkända former av for-loopar:

```
a int: 3;
for a, (< a 10), 1 ->
  ...
;

a int: 3;
for a, (< a 10) ->
  ...
;

a int: 3;
for a, (< a 10), a ->
  ...
;
```

Som nämndes ovan är inkrementeringsvärdet valbart och kommer, om det inte anges explicit, att sättas till 1. Notera även i den sista for-loopen att inkrementeringsvärdet kan sättas till en variabel.

2.3.9 While-loopar

Även när det gäller while-loopar stämmer dessa rätt väl överrens med hur dessa fungerar i andra språk, men då RIPPLE använder sig av prefixnotation för aritmetiska och logiska uttryck ser det lite annorlunda ut.

Exempel på while-loopar:

```
a int: 3;
while(< a 10) ->
  println a;
  a: (+ a 1);
;

a int: 3;
b int: 5;
while(! (= a b)) ->
  println a;
  a: (+ a 1);
;
```

2.3.10 If-satser

För att på ett enkelt sätt kontrollera flödet i koden är if-satsen ett av de viktigaste verktygen. En if-sats i RIPPLE består av tre delar, varav två är valbara. Dessa är *if*, *elsif* och *else*. Det enklaste sättet att demonstrera hur dessa fungerar är att ge ett kort exempel:

```
a int: 3;
if (> a 10) ->
  println "Stort nummer";
elsif (& (<= a 10) (> a 4)) ->
  println "Medelstort nummer";
else ->
  println "Litet nummer";
;
```

Om man vill definiera hur stora talen är på fler sätt kan man enkelt lägga in fler *elsif*-satser i koden. Likväl kan man om man bara har två fall plocka bort *elsif* helt:

```
a int: 3;
if (> a 10) ->
```

```
  println "Stort nummer";
else ->
  println "Litet nummer";
;
```

Skulle man bara vilja kontrollera om ett uttryck stämmer utan att göra något om så inte är fallet kan även *else* plockas bort ur if-satsen:

```
a int: 3;
if (> 10 a) ->
  print "10 är större än: ";
  println a;
;
```

2.3.11 Scope och skuggning av variabler

Ett scope kan ses som en container, eller en låda, som innehåller variabler, funktioner och styrstrukturer. Genom att använda sig av denna analogi kan man säga att de variabler som deklareras i ett scope kommer vara de som är närmast synliga för resten av scopet, då andra variabler kan vara deklarerade i lådan utanför. Detta är kanske enklast att visa med ett exempel:

```
defun init -> #ett scope
  x int: 3;
  y int: 3;
  for x, (< x 10), 1 -> #ett nytt scope
    y int: 4;
    println (+ x y);
  ;
;
```

Till synes deklareras ett x och ett y i init-funktionens scope. I for-loopen deklareras y om och därigenom kommer det y som är synligt för *println* att vara det som deklarerats i for-loopen.

2.3.12 Statisk bindning

Statisk bindning innebär att funktioner exekveras i den kontext de deklarerats och inte i den kontext varifrån de anropas. För att tydliggöra detta är följande exempel lämpligt:

```
defun init ->
  x int: 3;
  defun int multiply(int a) ->
    return (* x a);
  ;;
  defun do_multiplication ->
    x int: 4;
    return multiply(x);
  ;;
  do_multiplication();
;
```

Som exemplet visar deklareras ett x i init-funktionen och ett annat x i funktionen *do_ multiplication*. Hade dynamisk bindning använts skulle det x som används i funktionen *multiply* ha varit det som deklareras i funktionen *do_ multiplication*. På grund av den statiska bindningen kommer det vara det x som deklarerats i init-funktionen som används.

2.3.13 Rekursion

Funktioner i RIPPLE kan användas rekursivt för att lösa problem. Det klassiska exemplet på rekursion är hur man räknar ett tals fakultet. I RIPPLE ser koden för denna funktion ut som följer:

```
defun int nFak(int n) ->
  if (<= n 0) ->
    return 1;
  else ->
    return (* n nFak((- n 1)));
  ;
;
```

Detta innebär att funktionen antingen returnerar 1 till den som anropar den, eller värdet av funktionsanropet till nFak med parametern n-1 gånger n. På så sätt skapas en rekursiv kedja där man för varje steg minskar n med 1 och anropar funktionen om och om igen så länge som inparametern inte är mindre eller lika med 0. Likväl skulle man även kunna tänka sig en funktion som beräknar det n:te fibonaccitalet:

```
defun int fib(int n) ->
  if (<= n 1) ->
    return 1;
```

```
  else ->
    return (+ fib((- n 1)) fib((- n 2)));
  ;
;
```

2.3.14 Exempel på program

Ett fullständigt program där man drar nytta av de kodexempel som tidigare givits för att visa alla fibonaccital upp till ett visst stopp:

```
defun int fib(int n) ->
  if (<= n 1) ->
    return 1;
  else ->
    return (+ fib((- n 1)) fib((- n 2)));
  ;
;

defun init ->
  i int: 0;
  for i, (< n 10), 1 ->
    println fib(i);
  ;
;
```

Om man skulle vilja summera alla tal från 1 till n skulle man kunna använda sig av följande kod:

```
#En iterativ version
defun int sum_i(int n) ->
  i int: 1;
  s int: 0;
  for i, (<= i n), 1 ->
    s: (+ s i);
  ;
  return s;
;

#En rekursiv variant
defun int sum_r(int n) ->
  if (<= n 0) ->
    return 0;
```

```
  else ->
    return (+ n sum_r((- n 1)));
  ;
;

defun init ->
  print "Summan av alla tal från 1 till 5: ";
  println sum_i(5);
  println sum_r(5);
;
```

Ett ganska vanlig uppgift man ställs inför som nybörjarprogrammerare är hur man skriver ut multiplikationstabeller upp till ett givet tal. Detta kräver att man nästlar loopar och även detta är något som stöds i RIPPLE.

```
defun void mult_table(int n) ->
  i int: 1;
  for i, (<= i n) ->
    j int: i;
    while (<= j n) -> #While-loop nästlad i for-loopen
      print (* i j);
      print " ";
      j: (+ j 1);
    ;
    println;
  ;
;

defun init ->
  mult_table(10);
;
```

3 Systemdokumentation

I systemdokumentationen ges en mer teknisk överblick över systemet och hur det fungerar. Detta innefattar till exempel hur systemet går från kod till exekvering, vad interpretatorn har för klasser den använder sig av, samt grammatiken för språket.

3.1 Beskrivning av språket

RIPPLE är i stor utsträckning en blandning mellan egenskaper jag tycker är bra hos olika språk i kombination med en egen syntax. Ambitionen med språket var att det skulle vara typat och ha tydliga strukturer vad det gäller funktioner, statements och placering av kod. Typningen fungerar i någon utsträckning men inte fullt ut. Målet med att ha tydliga strukturer för hur man placerar kod och så vidare kommer som en reaktion på vissa språks totala avsaknad av detsamma. Genom att se till att all kod skrivs i funktioner slipper man det virrvarr av uttryck lite här och där i källkodsfilerna vilket kan göra programmet svårläsligt eller svårförståeligt. Precis som i språk som C och C++ tvingar RIPPLE fram en specificerad ingångspunkt i programmet varifrån resten av programmet körs.

3.2 Från källkod till exekvering

För att kunna köra ett program skrivet i RIPPLE måste systemet hantera ett flertal olika steg. Det första steget går ut på att parsa källkodsfilen för att skapa funktionsdefinitioner. Dessa funktionsdefinitioner består av olika *structs* beroende på vilka statements som identifierats. För varje funktion läggs dessa statements i en lista i den ordning de parsats, vilket ger en sekventiell struktur för exekvering av varje funktion. I steg två skickas dessa funktionsdefinitioner till interpretatorn vilken i sin tur skapar ett topp-scope, eller en container, där alla funktioner sparas. När detta gjorts kontrollerar interpretatorn att det finns en funktion vid namn *init*. Detta görs för att funktionen *init* är ingångspunkten för alla program skrivna i RIPPLE. Om det inte finns en sådan funktion ges ett felmeddelande och interpretatorn avslutas. Om systemet kommit så här långt startas exekveringen genom att containerklassens run-metod anropas.

3.2.1 Beståndsdelar

Exekveringssystemet för RIPPLE består i grunden av två distinkta delar: parsern och interpretatorn. Ingången för all hantering av RIPPLE-kod är

interpretatorn som beroende på kontext utnyttjar parsern på olika sätt. Det ena fallet är när man använder den interaktiva tolken och det andra när man anger en fil interpretatorn skall exekvera. Parsern genomför först en lexikalisk analys där den försöker skapa tokens av den inkommande strömmen tecken. Dessa kan till exempel bilda textsträngar, kommentarer och identifierare för att bara nämna några. Nästa steg är matchningen av dessa tokens mot de regler som satts upp för de olika saker språket innehåller som funktionsdefinitioner, tilldelningssatser och matematiska uttryck. Eftersom att de grundläggande behållarna i språket är funktioner kommer resultatet av parsningen vara en lista med funktionsdefinitioner i den ordning de skrivits in i källkoden. Dessa funktionsdefinitioner innehåller i sig alla satser som skall utföras. Parsern avslutar med att returnera den lista med funktionsdefinitioner till den som anropat parsefunktionen, vilket är interpretatorn. Denna fortsätter sedan med att steg för steg exekvera de instruktioner den fått.

I praktiken består alltså systemet av en ”förberedande” del och en exekverande del, där den förberedande delen tar hand om den råa bearbetningen av källkoden medan den exekverande delen omvandlar instruktionerna den fått från parsern till något faktiskt resultat.

3.2.2 Klasser

Under interpreteringsfasen av körningen använder sig RIPPLE av tre huvudklasser och tre stödklasser. Huvudklasserna är *Program*, *Function* och *Scope*. Stödklasserna är *CommonMethods*, *MathNode* och *LogNode*.

Program är den klass vari alla funktioner som definierats på grundnivå huserar. Det är här funktioner letas upp och returneras till anroparen, om funktionerna finns definierade på grundnivå det vill säga.

Function är den klass som hanterar all grundläggande funktionalitet för def funktioner som definierats. De objekt som instansieras som *Function* måste inte av nödvändighet definieras på grundnivå utan kan definieras inuti en funktion eller ett *Scope*. En funktion hanterar ett eller flera statements.

Scope är en klass som hanterar ett enda statement, vilket är av typerna *WhileStmt*, *ForStmt* eller *IfStmt*. Det som dock bör noteras är att dessa statements i sig kan innehålla funktionsdefinitioner och scope-bara statements likväl som bara enkla statements.

Det som är gemensamt för dessa klasser är att de i instansieringsfasen automatiskt får en pekare till den kontext de befinner sig i. Detta är alltså inte en dynamisk bindning utan en statisk sådan. Genom att man skickar med denna pekare får man ett enkelt sätt att leta upp variabler i den omgivande miljön och får även lätt möjligheten att evaluera uttryck, givet en viss kontext.

Som namnet antyder är stödklasserna till för att hjälpa interpretatorn med olika uppgifter under exekveringen. Klassen *CommonMethods* är ett sätt att korta ner källkoden för interpretatorn genom att samla ihop metoder som delas mellan klasserna *Function* och *Scope*. För att kunna göra detta får dessa vid instansiering en pekare till en instans av klassen *CommonMethods*. Detta gör att man i de andra klasserna inte behöver skapa så många metoder utan kan använda sig av Rubys inbyggda *method_missing* metod för att genom den anropa rätt funktion i *CommonMethods*-objektet.

Anledningen till att klasserna *MathNode* och *LogNode* finns är att alla aritmetiska och logiska uttryck i RIPPLE skrivs i prefixnotation och Ruby använder infix. Dessa klasser används alltså för att evaluera uttryck i prefixnotation på ett sätt som Ruby förstår. Detta går i korta drag ut på att man för varje uttryck i prefix skapar en nod av rätt typ, vilken i sin tur skapar en ”vänsternod” och en ”högernod” och på detta sätt skapas en trädstruktur med uttrycket i infix. När man sedan traverserar trädet ”in order” och evaluerar uttrycken under tiden får man till slut resultatet av infixuttrycket.

3.3 Kodstandard

Kodstandard i RIPPLE är ett väldigt löst begrepp. De enda riktiga reglerna för hur koden får se ut är namn på funktioner och variabler, samt var man får placera kod. Namn på variabler och funktioner måste börja på stor eller liten bokstav och får sedan följas av ett godtyckligt antal bokstäver och tecknet ”_”.

När det gäller hur kod får skrivas gäller regeln att all kod måste vara placerad inom funktionsdefinitioner, det vill säga inga statements utanför en funktionskropp. I övrigt när det gäller mellanslag, TAB, radbrytning och så vidare är det mesta godkänt. Det enda kravet som ställs är att nyckelord och namn på funktioner och variabler är sammanhängande.

3.4 Projektfilstruktur

I den tarboll systemet distribueras i ser strukturen ut som följande:

```
doc/
   grammar.txt
   RIPPLE.pdf
emacs/
   ripple-mode.el
interpreter/
   lib/
```

```
      function_expression_handler.rb
      math_logic.rb
      parser.rb
      rdparse.rb
   ripple.rb
README.txt
```

3.5 Grammatik

```
1  <program>      ::= <func_def>+
2
3  <func_def>     ::= "defun" <type> <identifier> "("
4                        <param_list> ")" "->"
5                             <stmt_list> <terminator>
6                    | "defun" <identifier> "("
                       <param_list> ")" "->"
7                             <stmt_list> <terminator>
8                    | "defun" <type> <identifier> "("
                       ")" "->"
9                             <stmt_list> <terminator>
10                   | "defun" <type> <identifier> "("
                       ")" "->"
11                            <stmt_list> <terminator>
12                   | "defun" <type> <identifier> "->"
13                            <stmt_list> <terminator>
14                   | "defun" <identifier> "->"
15                            <stmt_list> <terminator>
16
17 <func_call>    ::= <identifier> "(" <arg_list> ")"
18
19 <terminator>   ::= ";"
20
21 <stmt_list>    ::= <stmt>
22                   | <stmt_list> <stmt>
23
24 <stmt>         ::= (<func_def>
25                   | <decl_stmt>
26                   | <assign_stmt>
27                   | <io_stmt>
28                   | <sel_stmt>
```

```
29                    | <iter_stmt>
30                    | <break_stmt>
31                    | <return_stmt>) <terminator>
32
33  <decl_stmt>    ::= <identifier> <type>
34
35  <assign_stmt>  ::= <identifier> <type> ":"
36                       (<expression> | <read_stmt>)
37                    | <identifier> : (<expression> |
                        <read_stmt>)
38
39  <io_stmt>      ::= <print_stmt> | <read_stmt>
40
41  <print_stmt>   ::= ("println" | "print") "("
42                              <expression>
43                              | <identifier>
44                              | <literal> ")"
45                    | "prinln"
46                    | "printf" "(" <expression> ","
                        <string> ")"
47
48  <read_stmt>    ::= "read(" (stdin|<string>) ","
49                              <identifier> ")"
50
51  <sel_stmt>     ::= <if_stmt>
52
53  <if_stmt>      ::= "if" <expression> "->"
54                           <stmt>+
55                       ("elsif" <expression> "->"
                          <stmt>+ )*
56                       ("else" "->" <stmt>+ )*
57
58  <iter_stmt>    ::= <while_stmt> | <for_stmt>
59
60  <while_stmt>   ::= "while" <pred_expr> "->"
61                      <stmt>+
62
63  <for_stmt>     ::= "for" <identifier> ","
64                      <pred_expr> "," <numeric> "->"
65                           <stmt>+
```

```
66                      | "for" <identifier> "," <pred_expr>
                          "," <identifier> "->"
67                            <stmt>+
68                      | "for" <identifier> "," <pred_expr>
                          "->"
69                            <stmt>+
70
71  <break_stmt>   ::= "break"
72
73  <return_stmt>  ::= "return" <expression>
74
75  <param_list>   ::= <type> <identifier>
76                      | <param_list> "," <type>
                          <identifier>
77
78  <arg_list>     ::= <expression>
79                      | arg_list "," <expression>
80
81  <expression>   ::= ("(" (<aritm_expr> | <pred_expr>)
82                          ")"
83                      | <identifier>
84                      | <func_call>
85                      | <literal>)
86
87  <aritm_expr>   ::= <aritm_oper> (<numeric>
88                                   | <identifier>
89                                   | <expression>){2}
90
91  <pred_expr>    ::= (<rel_oper> | <log_oper>)
92                       <expression> <expression>
93
94  <aritm_oper>   ::= "+"
95                      | "-"
96                      | "*"
97                      | "/"
98                      | "^"
99                      | "%"
100
101 <log_oper>     ::= "!"
102                     | "&"
103                     | "|"
```

```
104                     | "="
105
106 <rel_oper>      ::= "<"
107                     | ">"
108                     | "<="
109                     | ">="
110
111 <identifier>   ::= [a-zA-Z][a-zA-Z_]*
112
113 <literal>      ::= <numeric> | <string>
114
115 <numeric>      ::= <int> | <float>
116
117 <int>          ::= [0-9]+
118
119 <float>        ::= [0-9]+"."[0-9]+
120
121 <bool>         ::= true|false
122
123 <string>       ::= " sequence of chars "
124
125 <container>    ::= "list" | "list<" <type> ">"
126
127 <type>         ::= "void"
128                     |"int"
129                     | "float"
130                     | "bool"
131                     | "string"
```

4 Programkod

I denna avdelning presenteras den kod som implementerats för att realisera programmeringsspråket RIPPLE. Den kod som inte redovisas är den för den underliggande parsern, Rdparse. Anledningen till att denna inte redovisas är jag inte skrivit den och jag således inte kan ta åt mig äran för den. Den som dock är intresserad av att se koden för rdparsern finns denna att beskåda i tarbollen för RIPPLE.

4.1 Parsern

```
1  #!/usr/bin/env ruby
2  # -*- coding: utf-8 -*-
3
4  require './lib/rdparse.rb'
5
6  class FuncDef
7    attr_accessor :type, :identifier, :param_list, :stmt_list
8    def initialize(type, identifier, param_list, stmt_list)
9      @type, @identifier, @param_list, @stmt_list =
10       [type, identifier, param_list, stmt_list]
11   end
12
13   def to_a
14     retarr = []
15     retarr << identifier
16     retarr << type
17     if param_list.length > 0 then
18       retarr << param_list
19     else
20       retarr << []
21     end
22     stmt_list.each do |line|
23       retarr << line
24     end
25     retarr
26   end
27 end
28
29 FunctionParam = Struct.new(:type, :identifier)
30 DeclStmt = Struct.new(:identifier, :type)
```

```
31 AssignStmt = Struct.new(:identifier, :type, :value)
32 ReassignStmt = Struct.new(:identifier, :expression)
33
34 ReadStmt = Struct.new(:source, :target)
35 PrintStmt = Struct.new(:source)
36 PrintLnStmt = Struct.new(:source)
37 PrintfStmt = Struct.new(:source, :target)
38
39 ReturnStmt = Struct.new(:expression)
40 BreakStmt = Struct.new(:expression)
41
42 WhileStmt = Struct.new(:condition, :statements)
43 ForStmt = Struct.new(:identifier, :condition, :inc_value, :statements)
44
45 AritmExpr = Struct.new(:operator, :operand, :operand1)
46 NpredExpr = Struct.new(:operator, :operand)
47 PredExpr = Struct.new(:operator, :operand, :operand1)
48 RelPredExpr = Struct.new(:operator, :operand, :operand1)
49
50 IfStmt = Struct.new(:condition, :statements, :elsif_list, :else)
51 Elsif = Struct.new(:condition, :statements)
52 Else = Struct.new(:statements)
53
54 FuncCall = Struct.new(:name, :arguments)
55
56 Identifier = Struct.new(:name)
57
58 Booltype = Struct.new(:name)
59
60 Sstring = Struct.new(:value)
61 Numeric = Struct.new(:value)
62
63 class LangParser
64
65   def initialize
66     @langParser = Parser.new("Language Parser") do
67
68       token(/^#.*$/)
69       token(/\n/)
70       token(/\\n/)
71       token(/".*"/) {|s| s}
```

```
72        token(/\s+/)
73        token(/(\-\d+|\d+)\.\d+/) {|d| d.to_f}
74        token(/(\-\d+|\d+)/) {|d| d.to_i}
75        token(/[\w+\.*]+/){|m| m}
76        token(/\w+/){|m| m}
77        token(/\-\>/) {|m| m}
78        token(/(\<\=|\>\=)/) {|m| m}
79        token(/./) {|m| m}
80
81        start :program do
82          match(:program, :func_def) {
83            |func_lst, func_def|
84            func_lst << func_def
85            func_lst
86          }
87          match(:func_def){
88            |func_def|
89            [func_def]
90          }
91        end
92
93        rule :func_def do
94          match('defun', :type, :identifier, '(', :param_list, ')', '->',
95                :stmt_list, :terminator) {
96            | _, type, identifier, _, param_list, _, _, stmt_list, _ |
97            FuncDef.new(type, identifier, param_list, stmt_list)
98          }
99          match('defun', :identifier, '(', :param_list, ')', '->',
100               :stmt_list, :terminator) {
101           | _, identifier, _, param_list, _, _, stmt_list, _ |
102           FuncDef.new('void', identifier, param_list, stmt_list)
103         }
104         match('defun', :type, :identifier, '(', ')', '->', :stmt_list,
105               :terminator) {
106           | _, type, identifier, _, _, _, stmt_list, _ |
107           FuncDef.new(type, identifier, [], stmt_list)
108         }
109         match('defun', :identifier, '(', ')', '->', :stmt_list,
110               :terminator) {
111           | _, identifier, _, _, _, stmt_list, _|
112           FuncDef.new('void', identifier, [], stmt_list)
```

```
113         }
114         match('defun', :type, :identifier, '->', :stmt_list,
115               :terminator) {
116           | _, type, identifier, _, stmt_list, _ |
117           FuncDef.new(type, identifier,[],stmt_list)
118         }
119         match('defun', :identifier, '->', :stmt_list, :terminator) {
120           | _, identifier, _, stmt_list, _ |
121           FuncDef.new('void', identifier, [], stmt_list)
122         }
123       end
124
125       rule :stmt_list do
126         match(:stmt_list, :stmt, :terminator) {
127           |stmt_list, stmt, _|
128           stmt_list << stmt
129         }
130         match(:stmt, :terminator) {
131           |stmt,_|
132           [stmt]
133         }
134       end
135
136       rule :stmt do
137         match(:func_def) {
138           |stmt|
139           stmt
140         }
141         match(:io_stmt) {
142           |stmt|
143           stmt
144         }
145         match(:func_call) {
146           |call|
147           call
148         }
149         match(:iter_stmt) { |stmt|
150           stmt
151         }
152         match(:assign_stmt) {
153           |stmt|
```

```
154           stmt
155         }
156         match(:decl_stmt) {
157           |stmt|
158           stmt
159         }
160         match(:selection_stmt) {
161           |stmt|
162           stmt
163         }
164         match(:return_stmt) {
165           |stmt|
166           stmt
167         }
168         match(:break_stmt) {
169           |stmt|
170           stmt
171         }
172       end
173
174       rule :io_stmt do
175         match(:print_stmt) {
176           |stmt|
177           stmt
178         }
179         match(:read_stmt) {
180           |stmt|
181           stmt
182         }
183       end
184
185       rule :read_stmt do
186         match('read', '(', 'stdin', ',', :identifier, ')'){
187           |_,_,_,_,identifier,_|
188           ReadStmt.new('stdin', identifier)
189         }
190         match('read', '(', :string, ',', :identifier, ')'){
191           |_,_,source,_,target,_|
192           ReadStmt.new(source,target)
193         }
194       end
```

```
195
196         rule :print_stmt do
197           match('println', :expression) {
198             |_, expr|
199             PrintLnStmt.new(expr)
200           }
201           match('println') {
202             |_|
203             PrintLnStmt.new(Sstring.new(""))
204           }
205           match('print', :expression){
206             |_,expr|
207             PrintStmt.new(expr)
208           }
209           match('printf', '(', :expression, ',',  :string, ')'){
210             |_, _, expression, _, filename, _ |
211             PrintfStmt.new(expression, filename)
212           }
213         end
214
215         rule :decl_stmt do
216           match(:identifier, :type){
217             | identifier, type |
218             DeclStmt.new(identifier, type)
219           }
220         end
221
222         rule :assign_stmt do
223           match(:identifier, :type, ':', :expression) {
224             | identifier, type, _, expression |
225             AssignStmt.new(identifier, type, expression)
226           }
227           match(:identifier, ':', :expression) {
228             | identifier, _ , expression |
229             ReassignStmt.new(identifier, expression)
230           }
231         end
232
233         rule :iter_stmt do
234           match(:while_stmt){|stmt| stmt}
235           match(:for_stmt){|stmt| stmt}
```

```
236       end
237
238       rule :while_stmt do
239         match('while', '(', :pred_expr, ')', '->', :stmt_list){
240           |_,_,expr,_,_,stmts|
241           WhileStmt.new(expr, stmts)
242         }
243       end
244
245       rule :for_stmt do
246         match('for', :identifier, ',', '(', :pred_expr, ')', ',',
247               :numeric, '->', :stmt_list){
248           |_,identifier,_,_,expr,_,_,inc,_,stmts|
249           ForStmt.new(identifier, expr, inc, stmts)
250         }
251         match('for', :identifier, ',', '(', :pred_expr, ')', ',',
252               :identifier, '->', :stmt_list){
253           |_,identifier,_,_,expr,_,_,inc,_,stmts|
254           ForStmt.new(identifier, expr, inc, stmts)
255         }
256         match('for', :identifier, ',', '(', :pred_expr, ')', '->',
257               :stmt_list){
258           |_,identifier,_,_,expr,_,_,stmts|
259           ForStmt.new(identifier, expr, Numeric.new(1), stmts)
260         }
261       end
262
263       rule :selection_stmt do
264         match(:if_stmt){|stmt| stmt}
265       end
266
267       rule :if_stmt do
268         match('if', '(', :pred_expr, ')', '->', :stmt_list, :elsif_list,
269               :else){
270           |_,_,expr,_,_,stmts,elsif_list,els|
271           IfStmt.new(expr,stmts,elsif_list,els)
272         }
273         match('if', '(', :pred_expr, ')', '->', :stmt_list, :elsif_list){
274           |_,_,expr,_,_,stmts,elsif_list|
275           IfStmt.new(expr,stmts,elsif_list,nil)
276         }
```

```
277           match('if', '(', :pred_expr, ')', '->', :stmt_list, :else){
278             |_,_,expr,_,_,stmts,els|
279             IfStmt.new(expr,stmts,nil,els)
280           }
281           match('if', '(', :pred_expr, ')', '->', :stmt_list){
282             |_,_,expr,_,_,stmts|
283             IfStmt.new(expr,stmts,nil,nil)
284           }
285         end
286
287         rule :elsif_list do
288           match(:elsif_list, :elsif) {
289             |elsif_list, els|
290             elsif_list << els
291           }
292           match(:elsif){|els| [els]}
293         end
294
295         rule :elsif do
296           match('elsif', '(', :pred_expr, ')', '->', :stmt_list){
297             |_,_,expr,_,_,stmts|
298             Elsif.new(expr,stmts)
299           }
300         end
301
302         rule :else do
303           match('else', '->', :stmt_list){
304             |_,_,stmts|
305             Else.new(stmts)
306           }
307         end
308
309         rule :return_stmt do
310           match('return', :expression){
311             |_, expr|
312             ReturnStmt.new(expr)
313           }
314         end
315
316         rule :break_stmt do
317           match('break'){
```

```
318           |_, _|
319           BreakStmt.new("Break")
320         }
321       end
322
323       rule :expression do
324         match('(', :aritm_expr, ')'){
325           |_,a_expr,_|
326           a_expr
327         }
328         match('(', :pred_expr, ')'){
329           |_,p_expr,_|
330           p_expr
331         }
332         match(:func_call){
333           |func_call|
334           func_call
335         }
336         match(:literal){
337           |literal|
338           literal
339         }
340         match(:identifier){
341           |identifier|
342           identifier
343         }
344       end
345
346       rule :aritm_expr do
347         match(/(\+|\-|\*|\/|\^|\%)/, :expression, :expression){
348           |op,exp,exp1|
349           AritmExpr.new(op, exp, exp1)
350         }
351       end
352
353       rule :pred_expr do
354         match('!', :expression){
355           |oper, expression|
356           NpredExpr.new(oper, expression)
357         }
358         match(:rel_oper, :expression, :expression){
```

```
359           |oper, expr, expr1|
360           RelPredExpr.new(oper, expr, expr1)
361         }
362         match(:log_oper, :expression, :expression){
363           |oper, expr, expr1|
364           PredExpr.new(oper, expr, expr1)
365         }
366       end
367
368       rule :log_oper do
369         match(/(\!|\&|\|)/){
370           |oper|
371           oper
372         }
373       end
374
375       rule :rel_oper do
376         match(/(\<|\>|\<\=|\>\=|\=)/){
377           |oper|
378           oper
379         }
380       end
381
382       rule :func_call do
383         match(:identifier, '(', :arg_list, ')') {
384           | identifier, _, args, _ |
385           FuncCall.new(identifier, args)
386         }
387         match(:identifier, '(', ')') {
388           | identifier, _, _|
389           FuncCall.new(identifier,[])
390         }
391       end
392
393       rule :param_list do
394         match(:param_list, ',', :parameter) {
395           |lst, _, parameter|
396           lst << parameter
397
398         }
399         match(:parameter) {
```

```
400           |param|
401           [param]
402         }
403       end
404
405       rule :parameter do
406         match(:type, :identifier) {
407           |type, identifier|
408           FunctionParam.new(type, identifier)
409         }
410       end
411
412       rule :arg_list do
413         match(:arg_list, ',', :expression) {
414           |lst, _, expression|
415           lst << expression
416         }
417         match(:expression){
418           |expression|
419           [expression]
420         }
421       end
422
423       rule :literal do
424         match(:numeric) {
425           |num|
426           num
427         }
428         match(:booltype) {
429           |booltype|
430           booltype
431         }
432         match(:string) {
433           |str|
434           str
435         }
436       end
437
438       rule :numeric do
439         match(:float) {
440           |f|
```

```
441           Numeric.new(f)
442         }
443         match(Fixnum) {
444           |i|
445           Numeric.new(i)
446         }
447       end
448
449       rule :type do
450         match('float') {
451           |f|
452           'Float'
453         }
454         match('int') {
455           |i|
456           'Fixnum'
457         }
458         match('string') {
459           |s|
460           'String'
461         }
462         match('void') {
463           |t|
464           t
465         }
466         match('bool') {
467           |b|
468           b
469         }
470       end
471
472       rule :identifier do
473         match(/[a-zA-Z]+[a-zA-Z_]*/) {
474           |i|
475           Identifier.new(i)
476         }
477       end
478
479       rule :float do
480         match(Float) {
481           |f|
```

```
482           f
483         }
484       end
485
486       rule :booltype do
487         match(/(true|false)/) {
488           |type|
489           Booltype.new(type)
490         }
491       end
492
493       rule :string do
494         match(/\".*\"/){
495           |s|
496           Sstring.new(s.to_s.gsub(/\"/,''))
497         }
498       end
499
500       rule :terminator do
501         match(';') {
502           |t|
503           t
504         }
505       end
506     end
507   end
508
509   def parse str
510     @langParser.parse str
511   end
512
513   def log(state = true)
514     @langParser.logger.level = state ? Logger::DEBUG : Logger::WARN
515   end
516
517 end
518
519 def parse_file filename
520   lp = LangParser.new
521   lp.log false
522   f = File.open(filename)
```

```
523   functions = lp.parse(f.read)
524   f.close
525   functions
526 end
527
528 def parse_string string
529   lp = LangParser.new
530   lp.log false
531   functions = lp.parse string
532   functions
533 end
```

4.2 Interpretatorn

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

=begin
An interpreter for the RIPPLE language.
=end

#Just to get rid of a pesky little warning..
$VERBOSE = nil

require './library/parser.rb'
require './library/math_logic.rb'
require './library/function_expression_handler.rb'


LANG_NAME = "RIPPLE"

#Set to true if debugging should be enabled
$debug = false

STDOUT.sync = true
STDIN.sync = true

#The statement types that should be scopeable
$scopeable = [:WhileStmt, :ForStmt, :IfStmt]

##
# This class can be seen as the container
# for all functions and it is this class
# that handle all interactions between
# different functions defined at the
# top level.
##
class Program
  attr_accessor :functions

  def initialize
    @functions = {}
  end
```

```
40
41    def get_function(function_name)
42      puts "in get_function container" if $debug
43      if function_name.class == Array then
44        function_name = function_name.join()
45      end
46      print "#{function_name}\n" if $debug
47      if(functions.key?(function_name))
48        Function.new(functions[function_name],self)
49      end
50    end
51
52    def has_function(function_name)
53      puts "in has function (container)" if $debug
54      print "#{function_name}\n" if $debug
55      puts function_name.class if $debug
56      if function_name.class == Array then
57        puts "function_name.class == Array" if $debug
58        function_name = function_name.join()
59      end
60      print "got function_name: #{function_name}\n" if $debug
61      return (functions[function_name] != nil)
62    end
63
64    def add_function(function)
65      if has_function(function.identifier.name) then
66        raise Exception, "Duplicate functions #{function.identifier.name}!"
67      end
68      functions[function.identifier.name] = function
69    end
70
71    def get_function_parameter_list(function_name)
72      functions[function_name].param_list
73    end
74
75    def print_functions
76      functions.each_key do |key|
77        puts key
78      end
79    end
80
```

```
81    def run
82      get_function("init").call
83    end
84
85  end
86
87  ##
88  # This class represents a Function. This is
89  # what a program in RIPPLE is made up of.
90  ##
91  class Function
92
93    attr_accessor :identifier, :parent, :return_type,
94    :parameter_list, :variables, :statements, :functions, :cm,
95    :variable_types
96
97    def initialize(function, parent)
98      @identifier = function.identifier.name
99      @parent = parent
100     @name = function.identifier
101     @variables = Hash.new
102     @variable_types = Hash.new
103     @cm = CommonMethods.new(self)
104
105     if($debug) then
106       puts function.param_list.class
107       function.param_list.each do |param|
108         puts param.class
109       end
110     end
111     @parameter_list = function.param_list
112
113     if(function.type != 'void') then
114       variables["_return_value_"] = nil
115     end
116     @statements = function.stmt_list
117     @functions = {}
118   end
119
120   def print_statements
121     statements.each do |statement|
```

```
122       puts statement
123     end
124   end
125
126   # This is the method that is called when
127   # the program wants to use the function
128   # and this is where all work is done.
129   def call(*args)
130     if($debug) then
131       print_variables
132       puts "In function #{identifier}"
133       puts "args: "
134       args.each do |arg|
135         print "#{arg},"
136       end
137       puts "statements:"
138       print_statements
139       print "\n"
140     end
141
142     if (args.length == 1) then
143       args = args[0]
144     end
145
146     if (args.length != parameter_list.length)
147       raise Exception, "Argument list length doesn't match" +
148         "parameter list length in call to function #{identifier}"
149     end
150
151     # Iterate through the parameter list
152     # to make sure that the provided
153     # arguments match.
154     parameter_list.each_index do |idx|
155       if (parameter_list[idx].type == "bool") then
156         if(args[idx].class.to_s != "FalseClass" &&
157            args[idx].class.to_s != "TrueClass") then
158           raise Exception, "Argument #{idx} not valid for function " +
159             "#{identifier}. Argument should be " +
160             "\"#{parameter_list[x].type}\"" +
161             ", was\ #{args[x].class}"
162         end
```

```
163       elsif (parameter_list[idx].type != args[idx].class.to_s)
164         raise Exception, "Argument #{x} not valid for function " +
165           "#{identifier}. Argument should be " +
166           "\"#{parameter_list[x].type}\", was #{args[x].class}"
167       end
168     end
169
170     parameter_list.each_index do |idx|
171       variables[parameter_list[idx].identifier.name] = args[idx]
172     end
173
174     # Iterate through the statements
175     # in the function and evaluate them.
176     statements.each do |statement|
177       if statement.class == ReturnStmt then
178         evaluate(statement, self)
179         break
180       elsif $scopeable.index("#{statement.class}".to_sym) != nil then
181         Scope.new(self, statement).run
182       else
183         puts "Should do something" if $debug
184         evaluate(statement, self)
185       end
186     end
187
188     print_variables if $debug
189
190     # If the function was defined with
191     # a return type then return the value
192     # that should be stored at "_return_value_"
193     if(variables.key?("_return_value_")) then
194       return variables["_return_value_"]
195     end
196   end
197
198   def method_missing(sym, *args)
199     puts "In method missing function, #{sym}" if $debug
200     instance_eval("cm.#{sym}(#{args})")
201   end
202
203 end
```

```
204
205 ##
206 # A Scope in RIPPLE is the "lowest"
207 # container type. It simply has one
208 # statement which it runs. The
209 # statement on the other hand can
210 # itself have many statements.
211 ##
212 class Scope
213
214   attr_accessor :parent, :statement, :variables, :cm, :functions,
215   :variable_types
216
217   def initialize(parent, statement)
218     @parent = parent
219     @statement = statement
220     @variables = Hash.new
221     @variable_types = Hash.new
222     @functions = Hash.new
223     @cm = CommonMethods.new(self)
224   end
225
226   def run
227     puts "In run" if $debug
228     evaluate(statement, self)
229     print_variables if $debug
230   end
231
232   def method_missing(sym, *args)
233     puts "In method missing scope" if $debug
234     self.instance_eval("cm.#{sym}(#{args})")
235   end
236
237 end
238
239 ##
240 # This class represents common methods
241 # that the classes Function and Scope
242 # might want to use. Every method in
243 # this class operates relative to the
244 # context specified when initializing.
```

```
245 ##
246 class CommonMethods
247
248   attr_accessor :context
249
250   # Initialize with the callers instance
251   def initialize(context)
252     @context = context
253   end
254
255   def create_variable(args)
256     name, value = args[0], args[1]
257     puts "In create variable" if $debug
258     if (context.variables.key?(name)) then
259       raise Exception, "Cannot declare the same variable " +
260         "(\"#{name}\") multiple times!"
261     else
262       value = context.get_correct_value(value, context.variable_types[name])
263       context.variables[name] = value
264     end
265   end
266
267   def get_correct_value(args)
268     value, type = args[0], args[1]
269
270     if(type == "Int")
271       value = value.to_i
272     elsif(type == "Float")
273       value = value.to_f
274     elsif(type == "String")
275       value = value.to_s
276     elsif(type == "Bool")
277       if value == "true"
278         value = true
279       else
280         value = false
281       end
282     end
283     return value
284   end
285
```

```
286   def set_variable_type(args)
287     name, type = args[0], args[1]
288     context.variable_types[name] = type
289   end
290
291   def set_variable(args)
292     name, value = args[0], args[1]
293     if (context.variables.key?(name)) then
294       type = context.variable_types[name]
295       value = context.get_correct_value(value, type)
296       context.variables[name] = value
297     else
298       context.parent.set_variable(name, value)
299     end
300   end
301
302   def unset_variable(args)
303     name = args[0]
304     context.variables.delete_if { |key,val|
305       key == name
306     }
307   end
308
309   def get_variable(args)
310     puts "in get variable" if $debug
311     puts "args = #{args}" if $debug
312     name = args
313     if args.class == Array then
314       name = args.join()
315     end
316     if (context.variables.key?(name)) then
317       if (context.variables[name] == nil) then
318         raise Exception, "Variable #{name} uninitialized!" +
319           " Cannot be used this way"
320       else
321         context.variables[name]
322       end
323     else
324       puts "Did not find variable #{name}" if $debug
325       context.parent.get_variable(name)
326     end
```

```
327   end
328
329   def print_variables(args)
330     puts "Variables at end of #{context.class}" if $debug
331     context.variables.each_pair do |key, val|
332       puts "#{key} = #{val}"
333     end
334   end
335
336   def has_function(function_name)
337     if (function_name.class == Array) then
338       function_name = function_name.join()
339     end
340     if (context.functions[function_name] != nil) then
341       return true
342     else
343       return context.parent.has_function(function_name)
344     end
345   end
346
347   def get_function(function_name)
348     if (function_name.class == Array) then
349       function_name = function_name.join()
350     end
351     if (context.functions[function_name] != nil) then
352       return Function.new(context.functions[function_name], context)
353     else
354       return context.parent.get_function(function_name)
355     end
356   end
357
358   # This is the real work horse of this class.
359   # It simply takes an argument and decides
360   # what to do with it.
361   def eval(arg)
362     puts "In eval" if $debug
363     if arg.class == Array
364       arg = arg.join()
365     end
366     instance_eval(arg)
367   end
```

```
368
369   def method_missing(sym, *args)
370     self.instance_eval("context.parent.#{sym}(#{args})")
371   end
372
373 end
374
375
376 ##
377 # Main evaluation loop. This determines
378 # what kind of statement that is passed
379 # to the method and takes action according
380 # to that.
381 #
382 # All calls made in this method is to
383 # methods defined in:
384 # function_expression_handler.rb
385 ##
386 def evaluate(stmt, context)
387
388   if stmt.class == PrintStmt
389     result = evaluate(stmt.source, context)
390     context.eval("print \"#{result}\"")
391   elsif stmt.class == PrintLnStmt
392     result = evaluate(stmt.source, context)
393     context.eval("puts \"#{result}\"")
394   elsif stmt.class == AssignStmt
395     context.set_variable_type(stmt.identifier.name, stmt.type)
396     context.create_variable(stmt.identifier.name,
397                             get_expression(stmt.value, context))
398   elsif stmt.class == ReassignStmt
399     context.set_variable(stmt.identifier.name,
400                          get_expression(stmt.expression, context))
401   elsif stmt.class == DeclStmt
402     context.set_variable_type(stmt.identifier.name, stmt.type)
403     context.create_variable(stmt.identifier.name, nil)
404   elsif stmt.class == FuncCall
405     handle_FuncCall(stmt, context)
406   elsif stmt.class == FuncDef
407     context.functions[stmt.identifier.name] = stmt
408   elsif stmt.class == ReturnStmt
```

```
409       context.set_variable("_return_value_",
410                            get_expression(stmt.expression, context))
411     elsif stmt.class == ForStmt
412       handle_ForStmt(stmt, context)
413     elsif stmt.class == WhileStmt
414       handle_WhileStmt(stmt, context)
415     elsif stmt.class == IfStmt
416       handle_IfStmt(stmt, context)
417     elsif stmt.class == ReadStmt
418       handle_ReadStmt(stmt, context)
419     elsif stmt.class == PrintfStmt
420       handle_PrintfStmt(stmt, context)
421     elsif stmt.class == PredExpr
422       get_expression(stmt, context)
423     elsif stmt.class == RelPredExpr
424       get_expression(stmt, context)
425     elsif stmt.class == NpredExpr
426       get_expression(stmt, context)
427     elsif stmt.class == AritmExpr
428       get_expression(stmt, context)
429     elsif stmt.class == Identifier
430       context.get_variable("#{stmt.name}")
431     elsif stmt.class == Sstring
432       stmt.value
433     elsif stmt.class == Numeric
434       stmt.value
435     elsif stmt.class == BreakStmt
436       "Break"
437     end
438
439 end
440
441 ##
442 # The method that actually sets up the
443 # interpreting environment and runs the
444 # code.
445 ##
446 def interpreter(instructions)
447   if instructions.nil? and instructions.length <= 0 then
448     raise Exception, "Got no instructions"
449   end
```

```
450
451   # Check for init function. No init = no program.
452   has_init = false
453   instructions.each do |func|
454     puts func.identifier.name if $debug
455     if func.identifier.name == "init"
456       has_init = true
457       break
458     end
459   end
460
461   if !has_init then
462     raise Exception, "No init function found!"
463   else
464
465     # Add all functions to the program and run the init function.
466     program = Program.new
467     instructions.each do |function|
468       program.add_function(function)
469     end
470     program.run
471   end
472
473 end
474
475 ##
476 # Display a friendly welcome message upon
477 # start of interpreter.
478 ##
479 def welcome
480   print "####\n" +
481     "## Welcome to a partly working interpreter " +
482     "for the #{LANG_NAME} language\n" +
483     "## If you feel lost please do \"help\"\n" +
484     "####\n\n"
485 end
486
487 ##
488 # Prints a message with help regarding how
489 # to use the interactive console.
490 ##
```

```
491 def help
492   print "####\n" +
493     "## Some general guidance regarding the usage of this interpreter\n" +
494     "## Useful commands:\n" +
495     "## exit: exits the interpreter\n" +
496     "## help: shows this help\n" +
497     "## pb: prints the current buffer\n" +
498     "## cb: clears the current buffer\n" +
499     "## rm <num>: removes the chosen line from the buffer\n" +
500     "## md <num> <line>: modifies the chosen line in the buffer\n" +
501     "## ins <num> <line>: inserts specified line at the specified" +
502     " number in buffer\n" +
503     "## run: try to execute the buffer *fingers crossed*\n" +
504     "####\n\n"
505 end
506
507 ##
508 # A nice way to display the error message. This
509 # will be used for all exceptions that occur
510 # inside the interpreter or indeed all files
511 # that is used by the interpreter.
512 ##
513 def display_error(errmsg)
514     puts
515     puts "###"
516     puts "An error occurred:"
517     puts errmsg.message
518     puts "###"
519     puts
520 end
521
522 ##
523 # The code that is run when the user wants
524 # to use the interactive version of the
525 # interpreter. This basically takes some
526 # input and determines the action based
527 # on content.
528 ##
529 def interactive_mode(already_initialized=false)
530
531   if(!already_initialized)
```

```
532     welcome()
533     $buffer = []
534   end
535
536   #Make sure we can rescue if something goes wrong. ie typo...
537   begin
538     while true do
539       print "> "
540       command = gets.chomp
541       if command == "exit" || command == "quit"
542         break
543       elsif command == "help"
544         help()
545       elsif command == "debug"
546         $debug = !$debug
547         puts "Debug -> #{$debug}"
548       elsif command == "pb" #Print buffer
549         puts "Contents of buffer:"
550         Range.new(0,$buffer.length).each do |linum|
551           puts "#{linum}: #{$buffer[linum]}"
552         end
553       elsif command == "cb" #Clear buffer
554         puts "Clearing buffer..."
555         $buffer = Array.new
556       elsif command =~ /^md\s*\d+.*/ #Modify line of buffer
557         $buffer[command.scan(/\d+/)[0].to_i] =
558           command.sub(/md\s*\d+/,'')
559       elsif command =~ /^rm\s*\d+$/ #Remove line from buffer
560         $buffer.delete_at(command.scan(/\d+/)[0].to_i)
561       elsif command =~ /^ins\s*\d+.*/ #Insert line in buffer
562         $buffer.insert(command.scan(/\d+/)[0].to_i,
563                        command.sub(/ins\s*\d+\s*/, ''))
564       elsif command == "run"
565         # Assemble instructions, parse and run
566         instructions = $buffer.join("\n")
567         instructions = parse_string instructions
568         interpreter instructions
569       else
570         $buffer.push command
571       end
572     end
```

```
573   rescue Exception => e
574     display_error(e)
575     interactive_mode(true)
576   end
577
578 end
579
580 ##
581 # Entry point when running the file. If a
582 # filename is given it will send the file
583 # to the parser and then run the interpreter
584 # with the instructions from the parser,
585 # otherwise it will start the interactive
586 # mode.
587 ##
588 begin
589   if ARGV.length == 0 then #Assume full interactive mode
590     interactive_mode(false)
591   else #Assume filename given
592     if !File.exists? ARGV[0].to_s then #Check that file exists
593       puts "The specified file \"#{ARGV[0].to_s}\" " +
594         "doesn't seem to exist."
595     else
596       if ARGV.length == 2
597         if ARGV[1] == "true"
598           $debug = true
599         end
600       end
601       instructions = parse_file(ARGV[0].to_s)
602       interpreter instructions
603     end
604   end
605 rescue Exception => e
606   display_error(e)
607 end
```

4.3 Funktioner och uttryck

```
# -*- coding: utf-8 -*-

##
# This method takes an expression
# and decides what to do with it
# based on the expressions class.
##
def get_expression (expression, context=nil)
  if expression.class == Sstring
    "\"#{expression.value}\""
  elsif expression.class == Numeric
    expression.value
  elsif expression.class == Identifier
    context.get_variable("#{expression.name}")
  elsif expression.class == Booltype
    expression.name
  elsif expression.class == AritmExpr
    infix = convert_aritm_to_infix(expression, context)
    infix
  elsif expression.class == PredExpr ||
      expression.class == RelPredExpr ||
      expression.class == NpredExpr
    infix = convert_logic_to_infix(expression, context)
    infix
  elsif expression.class == FuncCall
    handle_FuncCall(expression, context)
  end
end


###
# Handler for if statements. This function
# is used to iterate in a logical way through
# all steps involved in an if statement.
###
def handle_IfStmt(statement, context)

  puts statement if $debug

```

```
40    value = nil
41
42    if (evaluate(statement.condition, context) == true) then
43      puts "Condition true" if $debug
44      statement.statements.each do |stm|
45        value = evaluate(stm, context)
46      end
47    else
48      done = false
49      if (statement.elsif_list != nil) then
50        statement.elsif_list.each do |els|
51          break if (done == true)
52          if (evaluate(els.condition, context) == true) then
53            done = true
54            els.statements.each do |estm|
55              value = evaluate(estm, context)
56            end
57          end
58        end
59      end
60      if (done == false && statement.else != nil) then
61        statement.else.statements.each do |elstm|
62          value = evaluate(elstm, context)
63        end
64      end
65    end
66    value
67  end
68
69  ###
70  # This method takes care of while statements.
71  # This is done by iterating through each
72  # statement in the while statement and for
73  # each iteration check if the condition is
74  # true or false by evaluating the condition
75  # in the context of the source of the
76  # statement.
77  ###
78  def handle_WhileStmt(stmt, context)
79
80    puts stmt if $debug
```

```
81    introduced_vars = []
82
83    broken = false
84    while (evaluate(stmt.condition, context) == true && broken == false)
85      stmt.statements.each do |wstmt|
86        if wstmt.class == BreakStmt then
87          broken = true
88          break
89        end
90        value = evaluate(wstmt, context)
91        if value == "Break" then
92          broken = true
93          break
94        end
95        if wstmt.class == AssignStmt then
96          puts "assign stmt" if $debug
97          introduced_vars << wstmt.identifier.name
98        end
99      end
100     print "Introduced vars:\n#{introduced_vars}\n" if $debug
101     introduced_vars.each do |var|
102       context.unset_variable("#{var}")
103     end
104   end
105
106 end
107
108 ##
109 # Handler for for statements. Iterates through
110 # each statement of the for statement and
111 # for each iteration evaluates the condition.
112 ##
113 def handle_ForStmt(stmt, context)
114
115   identifier = stmt.identifier.name
116   condition = stmt.condition
117   increment = nil
118   broken = false
119
120   #Get the value with wich to increment with
121   if(stmt.inc_value.class == Identifier) then
```

```
122       increment = stmt.inc_value.name
123       increment = context.get_variable("#{increment}")
124     elsif (stmt.inc_value.class == Numeric) then
125       increment = stmt.inc_value.value
126     end
127
128     # Iterate through each statement in the for loop
129     # while the condition is true, and then increment.
130     while(evaluate(condition, context) == true && broken == false)
131       stmt.statements.each do |fstmt|
132         if(fstmt.class == BreakStmt) then
133           broken = true
134           break
135         end
136         value = evaluate(fstmt, context)
137         if value == "Break" then
138           broken = true
139           break
140         end
141       end
142
143       old_value = context.get_variable("#{identifier}")
144       test_value = old_value + increment
145       context.set_variable("#{identifier}", test_value)
146
147       if(evaluate(condition, context) == false) then
148         context.set_variable("#{identifier}", old_value)
149         break
150       end
151     end
152
153 end
154
155 ##
156 # This method handles read statements.
157 # Depending of parameters content is read
158 # either from stdin or a file.
159 ##
160 def handle_ReadStmt(stmt, context)
161
162     source = stmt.source
```

```
163   target = stmt.target.name
164   read_text = ""
165
166   if(source == "stdin") then
167     # This is suspicious. I had to explicitly tell
168     # ruby that i wanted $stdin, should be default.
169     read_text = $stdin.gets
170   elsif (source.class == Sstring)
171     f = File.new(source.value)
172     read_text = f.read()
173     f.close()
174   end
175
176   read_text = read_text.gsub("\"", '\"')
177   context.set_variable("#{target}", read_text.chomp)
178
179 end
180
181 ##
182 # Method for handling printing to
183 # file. If the file exists it
184 # defaults to appending to the
185 # file, else it creates a new.
186 ##
187 def handle_PrintfStmt(stmt, context)
188
189   source = stmt.source.name
190   target = stmt.target.value
191   f = nil
192
193   if (File.exists?(target)) then
194     puts "file exists"
195     f = File.open(target, 'a')
196   else
197     puts "creating new file"
198     f = File.open(target, 'w')
199   end
200
201   f.puts(context.eval("get_variable(\"#{source}\")").gsub("\"", ""))
202   f.close()
203
```

```
204 end
205
206 ##
207 # Method for function calls. Since
208 # all functions are stored within the
209 # context of the calling object, it
210 # simply asks for a new instance of the
211 # function and returns the value of the
212 # executed function.
213 ##
214 def handle_FuncCall(stmt, context)
215
216   if context.has_function(stmt.name.name) then
217     arg_list = []
218     stmt.arguments.each do |arg|
219       arg_list << evaluate(arg, context)
220     end
221     context.get_function(stmt.name.name).call(arg_list)
222   else
223     raise Exception, "There's no function with name: \"#{stmt.name.name}\""
224   end
225
226 end
```

4.4 Matematik och logik

```
# -*- coding: utf-8 -*-

##
# This file contains functions to convert
# arithmetic and logic expressions in prefix
# notation to the corresponding ones in infix.
##

##
# This method receives an expression
# in prefix notation and returns
# a calculated value.
##
def convert_aritm_to_infix(expr, context)
  head = MathNode.new(expr.operator, expr.operand,
                      expr.operand1, context)
  result = head.collect
  result
end

##
# This class will, when instantiated, create
# a tree of nodes which contains all operantions
# that need to be done to calculate a value.
##
class MathNode

  attr_accessor :operator, :operand1, :operand2, :left,
  :right, :context

  def initialize(operator, operand1, operand2, context)
    @operator, @operand1, @operand2, @context = [operator, operand1,
                                                  operand2, context]
    left, right = [nil,nil]

    if(operand1.class == AritmExpr) then
      @left = MathNode.new(operand1.operator, operand1.operand,
                           operand1.operand1, context)
    else
```

```
40        @left = operand1
41      end
42
43      if(operand2.class == AritmExpr) then
44        @right = MathNode.new(operand2.operator, operand2.operand,
45                              operand2.operand1, context)
46      else
47        @right = operand2
48      end
49    end
50
51    # Since all mathematical operations contain
52    # two operands we first collect the left node
53    # and then put an operator and finally
54    # collect the right node.
55    def collect
56      retStr = ""
57      if(left != nil && left.class == MathNode) then
58        retStr += "#{left.collect}"
59      elsif(left.class == Numeric)
60        retStr += "#{left.value}"
61      elsif(left.class == Identifier)
62        retStr += context.get_variable("#{left.name}").to_s
63      else
64        retStr += "#{evaluate(left, context)}"
65      end
66
67      if(operator.to_s == "^")
68        retStr += "**"
69      else
70        retStr += operator.to_s
71      end
72
73      if(right != nil && right.class == MathNode) then
74        retStr += "#{right.collect}"
75      elsif(right.class == Numeric)
76        retStr += "#{right.value}"
77      elsif(right.class == Identifier)
78        retStr += context.get_variable("#{right.name}").to_s
79      else
80        retStr += "#{evaluate(right, context)}"
```

```
81      end
82
83      puts retStr if $debug
84      puts "#{eval(retStr)}" if $debug
85
86      #Evaluating resulting string to make sure we return a valid number
87      eval("#{retStr}")
88    end
89
90  end
91
92  ##
93  # This method receives a logic expression
94  # in prefix and returns a calculated value.
95  ##
96  def convert_logic_to_infix(expr, context)
97    if(expr.class != NpredExpr) then
98      head = LogNode.new(expr.operator, expr.operand,
99                         expr.operand1, context)
100   else
101     head = LogNode.new(expr.operator, expr.operand,
102                        nil, context)
103   end
104   result = head.collect
105   result
106 end
107
108 ##
109 # This class represents a node in a tree
110 # of logic expressions.
111 ##
112 class LogNode
113
114   attr_accessor :operator, :operand1, :operand2, :left,
115   :right, :context
116
117   def initialize(operator, operand1, operand2=nil, context)
118     @operator, @operand1, @operand2, @context = [operator, operand1,
119                                                  operand2, context]
120     @left,@right = [nil, nil]
121
```

```
122     if(operand1.class == RelPredExpr || operand1.class == PredExpr)
123       @left = LogNode.new(operand1.operator, operand1.operand,
124                           operand1.operand1, context)
125     elsif(operand1.class == NpredExpr)
126       @left = LogNode.new(operand1.operator, operand1.operand,
127                           nil, context)
128     else
129       @left = operand1
130     end
131
132     if(operand2.class == RelPredExpr || operand2.class == PredExpr)
133       @right = LogNode.new(operand2.operator, operand2.operand,
134                            operand2.operand1, context)
135     elsif(operand2.class == NpredExpr)
136       @right = LogNode.new(operand2.operator, operand2.operand,
137                            nil, context)
138     else
139       @right = operand2
140     end
141   end
142
143   # This method needs to handle two basic
144   # cases. Either this node is a logic
145   # expression with the operator "!",
146   # which means that there only is one
147   # operand, or else there are two
148   # operands.
149   def collect
150     retStr = ""
151
152     # The case with only one operand
153     if(self.operator == "!") then
154       retStr += self.operator.to_s
155       if (left.class == LogNode)
156         retStr += "#{left.collect}"
157       elsif (left.class == AritmExpr)
158         retStr += convert_aritm_to_infix(left, context).to_s
159       elsif (left.class == Numeric)
160         retStr += left.value.to_s
161       elsif (left.class == Identifier)
162         retStr += "get_variable(\"#{self.operand1.name.to_s}\")"
```

```
163       elsif (left.class == Booltype)
164         retStr += "#{self.operand1.name}"
165       else
166         retStr += "#{self.operand1}"
167       end
168
169     # The case with two operands.
170     else
171       if(left != nil && left.class == LogNode) then
172         retStr += "#{left.collect}"
173       elsif (left.class == AritmExpr)
174         retStr += convert_aritm_to_infix(left, context).to_s
175       elsif(left.class == Numeric && self.operator != "!")
176         retStr += left.value.to_s
177       elsif(left.class == Identifier && self.operator != "!")
178         retStr += "get_variable(\"#{left.name.to_s}\")"
179       elsif(left.class == Booltype && self.operator != "!")
180         p "sadf"
181         retStr += left.name.to_s
182       elsif(self.operator != "!")
183         retStr += "#{self.operand1}"
184       end
185
186       if(self.operator == "&")
187         retStr += "&&"
188       elsif(self.operator == "|")
189         retStr += "||"
190       elsif(self.operator == "=")
191         retStr += "=="
192       else
193         retStr += operator
194       end
195
196       if(right != nil && right.class == LogNode) then
197         retStr += "#{right.collect}"
198       elsif(self.class != NpredExpr) then
199         if(right.class == Numeric)
200           retStr += right.value.to_s
201         elsif(right.class == AritmExpr)
202           retStr += convert_aritm_to_infix(right, context).to_s
203         elsif(right.class == Identifier)
```

```
204           retStr += "get_variable(\"#{right.name.to_s}\")"
205         elsif(right.class == Booltype)
206           retStr += "#{right.name}"
207         end
208       end
209
210     end
211
212     puts "should eval: #{retStr}" if $debug
213     context.eval("(#{retStr})")
214   end
215
216 end
```

5 Reflektioner

Det första som slår mig så här i slutfasen av projektet är att uppgiften i fråga synnerligen inte var av en trivial karaktär. Det är naturligtvis inget fel i sig, men jag tror att man kanske ibland kan stirra sig blind på uppgiften utan att på ett bra och strukturerat sätt dela upp problemen som uppstår i mer lätthanterliga delar. För mig tror jag att den svåraste uppgiften låg i att faktiskt komma på ett språk jag skulle kunna tänka mig att själv använda; för det är där fokus hela tiden varit. Det må vara så att språket i sin nuvarande form kanske inte är så användbart, men själva programmeringsstilen det följer är i stort sett precis som jag vill ha det. När det gäller den ursprungliga språkspecifikationen i ljuset av vad som faktiskt blev implementerat får jag nog hävda att det inte egentligen går att utröna någon märkbar skillnad. Det enda som inte implementerades var funktionalitet för att hantera listor eller arrayer, men det ser jag inte som en direkt viktig del i språket heller, så ingen skada skedd på den fronten.

Något som varit väldigt lärorikt med detta projekt är att få en insyn i vad som måste ske i det fördolda för att ett program skall kunna realiseras från något så enkelt som vanlig text. Detta är ett typexempel på en sådan sak som man inte tänker på när man utvecklar program, men som onekligen troligtvis i framtiden kommer att finnas med en någonstans i bakhuvudet.

Det jag dock inte är nöjd med vad gäller realiseringen av språket är dess exekveringstid. För ”vanlig” programmering märks ingen direkt skillnad mot att köra till exempel Ruby, men när man börjar leka med rekursiva funktioner kan det ibland kännas lite långsamt. Själv tror jag att detta har att göra med det sätt jag implementerat interpretatorn genom att denna läser rena instruktioner i listform och för varje instruktion måste leta upp vad den skall göra för att komma vidare. Om jag istället hade implementerat alla former av uttryck och liknande som klasser som alla hade haft en metod ”eval” eller liknande hade detta troligtvis snabbat upp exekveringen genom att interpretatorn i det fallet inte behöver ta reda på vad som skall göras, utan den funktionaliteten hade i så fall hamnat i respektive klass. Detta är onekligen en viktig erfarenhet att ta med sig, men tyvärr en som jag erfor för sent för att kunna göra något åt problemet.

Det man nog ändå måste säga kan vara det som känns viktigast med kursen i fråga om lärdomar är att man inser att verkligheten inte är svart eller vit i fråga om hur språk fungerar, hur man kan tolka saker, hur semantiken som är glasklar för någon inte alls är uppenbar för en annan och så vidare. Detta är nog något som man inte normalt sett tänker på och det är säkert sådant man kommer att se när vi väl granskar varandras språk. Likväl har projektet fått en att börja ifrågasätta konstruktioner och funktionalitet i de

språk man kan sedan tidigare. Detta stämmer även i Ruby. När man har suttit och jagat en bugg i några timmar som egentligen inte borde manifesterat sig alls kan man börja tveka över hur det är tänkt att det skall fungera, speciellt när det inte finns någon dokumentation gällande den funktionalitet man letar efter. Sen får man ju kanske anse att just ett sånt här projekt är ganska idealiskt för att lära sig det språk man implementerar i. Detta genom att man aktivt måste leta fram information för att försöka lösa de väldigt praktiska problem man stöter på.

Språket som helhet tycker jag dock är relativt lyckat, men implementeringen är det som skulle behöva förändras för att faktiskt kunna göra något användbart av det. Likväl kanske det inte skulle vara hållbart att i verkliga världen utveckla nya språk i redan interpreterade miljöer då dessa i praktiken måste evaluera den kod som den egna interpretatorn skall köra och lägger på så sätt till ett extra steg som kanske egentligen är rätt onödigt. Om man nu skulle få för sig att utveckla ett eget språk på ett mer seriöst sätt kanske det skulle vara en bra idé att behålla syntaxen men byta exekveringsmiljö till en kompilerad så att man slipper ”mellanhanden”.