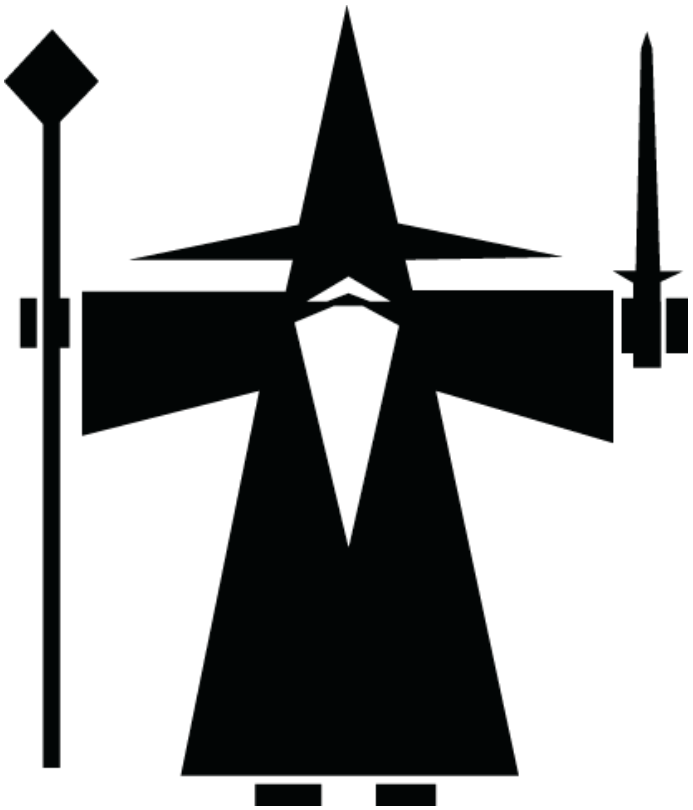


Gandalf

Projektdokumentation
TDP019 - Projekt: Datorspråk

Albin Englund, alben056@student.liu.se
Björn Jansson, bjoja869@student.liu.se

Sammanfattning

Vår uppgift i kursen TDP019 på programmet Innovativ Programmering på Linköpings universitet har varit att implementera ett eget programmeringsspråk. Detta språk har vi valt att kalla för Gandalf. Gandalf är ett objektorienterat språk som inspirerats utav Python samt Objective-C.

Gandalf är implementerat med hjälp utav ett verktyg som kallas RDparser som är skrivet i språket Ruby. Gandalf har många utav de vanligaste strukturerna som finns i de flesta programmeringsspråk nämligen; if-satser, for-loopar, while-loopar, funktioner och klasser. Även de mest grundläggande datatyperna som strängar, heltal och listor finns implementerade i Gandalf.

I denna rapport går vi igenom och redogör för hur Gandalf är uppbyggt och hur man skriver program i språket.

Innehållsförteckning

1. Inledning

Detta projekt genomfördes under första året på programmet Innovativ Programmering vid Linköpings universitet. Projektet utfördes under ramen för kursen "Projekt: Datorspråk" - TDP019 under andra terminen. Rapporten består utav tre avsnitt; användarhandledning, systemdokumentation, erfarenheter och reflektion.

1.1 Syfte

Syftet var att lära oss mer om hur ett programmeringsspråk är uppbyggt och vad som egentligen händer när programkod ska tolkas och exekveras. För att verkligen förstå detta på ett grundligt sätt så fick vi till uppgift att implementera vårt eget programmeringsspråk.

1.2 Idé

Vi hade en idé om att implementera ett språk som hade vissa syntaktiska likheter med Python och Objective-C. Vi ville göra språket helt indenteringsstyrt vilket tvingar användaren att skriva tydlig och mer läsbar kod. Tydlighet och läsbarhet är även applicerat gällande funktionsnamn som är utformade för att på ett lätt sätt kunna läsas från vänster till höger och ge ett lättförståeligt intryck.

Vi valde att göra ett interpreterat språk för att vi anser att det är lättare för en programmerare att testa kod och satser med en tolk snarare än att behöva kompilera om koden efter varje förändring som görs. Tolken ses som ett bra verktyg vid användning utav språket.

1.3 Implementeringsplan

Under den första tiden utav projektet hade vi en planeringsprocess. Under denna process pratade vi igenom olika idéer och tankar om hur vi ville att vårt slutgiltiga språk skulle se ut och fungera. Vi var båda ganska överens om vad vi ville arbeta mot och därför gick denna process väldigt smidigt.

Vi har under hela projektet jobbat tillsammans och har därför inte delat upp arbetet. Detta har väckt många bra diskussioner som vi båda anser har varit väldigt givande. Att frågasätta och bli frågasatt är viktigt för ett bra resultat.

Efter planeringsprocessen skrev vi grammatik för vårt språk. Till en början skrev vi denna i en enkel textfil för att ha en bra översikt. När vi ansåg att grammatiken var färdig så började vi implementera de syntaktiska regler som vi gjort för de enklare aritmetiska uttrycken. Detta byggde vi sedan vidare på.

1.4 Metod

För att implementera vårt språk har vi använt oss utav ett verktyg som heter "RDparser" (Recursive Descent parser). Detta är ett verktyg som är byggt i programmeringsspråket Ruby och används för att läsa och tolka text som är uppbyggd efter en viss grammatik/syntax. Verktyget hanterar tre olika delar utav implementeringsprocessen; lexer, tokenizer och parser. Dessa förklaras mer noga under systemdokumentationen, avsnitt 3.1.

2. Användarhandledning

Denna handledning är för dig som är intresserad utav att lära dig hur du kommer igång med, samt använder språket Gandalf.

2.1 Installation

Denna installationsguide utgår från att man använder Ubuntu. Följande installationsmoment måste utföras oberoende av vilket operativsystem man använder. Tillvägagångssättet för de olika installationerna skiljer sig dock mellan olika system. Den här guiden kommer bara förklara hur man går tillväga under operativsystemet Ubuntu.

2.1.1 Gandalf

Filerna som behövs för att använda Gandalf går att hitta på [länk] och är paketerade i en tarball-fil vid namn gandalf-1.0.tar. Paketet innehåller parser.rb, rdparser.rb, library.rb, gandalf.rb och tree.rb. Den innehåller även dokumentationen och grammatiken, vilka ligger i mappen Doc. De filer som ligger i mappen Doc är inte obligatoriska för att kunna köra språket Gandalf. Filstrukturen för paketet ser ut som följer:

```
/
     Doc/
          Example code/
               *.gdf
          grammar.txt
          Rapport.pdf
     gandalf.rb
     library.rb
     parser.rb
     rdparser.rb
     tree.rb
```

För att packa upp en tarball skriv följande kommandon i terminalen:

```
cd /sökväg/till/katalog/där/gandalf-1.0.tar/ligger
tar zxvf gandalf-1.0.tar
```

2.1.2 Ruby

Programmet är utvecklat och testat för Ruby 1.8.5 och man bör därför ha detta installerat. För att göra detta skriver man "`sudo apt-get install ruby`" i terminalen.

2.1.3 Interpretatorn

För att starta interpretatorn går du till den katalog där Gandalf.rb finns. Sedan skriver du:

```
ruby gandalf.rb
```

Vill du hellre köra en fil. Skriv följande:

```
ruby gandlaf.rb file.gdf
```

2.2 Komma igång

Innan vi börjar gå igenom språkets konstruktioner så är det bra att gå igenom hur interpretatorn fungerar. Antingen kan man starta interpretatorn utan argument, och då skriva koden direkt i prompten. När man vill köra den koden man skrivit in så trycker man enter utan att skriva något på raden. Vill man hellre köra en fil med programkod kan man ange sökvägen till denna fil som argument till interpretatorn. Tillvägagångssättet för att göra detta är uttryckt i avsnitt 2.1.3.

2.2.1 Ditt första program i Gandalf

Öppna en ny fil i en texteditor t.ex. Emacs, Vim eller TextMate. Här följer ett kodexempel för att skriva ut meningen "Hello world!" i språket Gandalf. Skriv följande rad i din nyskapade fil:

```
print "Hello world!"
```

Spara sedan denna fil som exempel1.gdf i samma katalog som du packat upp gandalf.rb. Kör sedan denna fil som beskrivet under 2.1.3.

2.2.2 Indentering

Gandalf är ett indenteringsstyrt språk vilket innebär att tabbar avgör var kodblock startar och slutar. Detta används främst för att koden ska bli så lättläst som möjligt men det uppfyller även fler funktioner. Bland annat tvingar det användaren utav språket att skriva snyggt strukturerad och prydlig kod då tabbar och radbrytningar är ett krav för att programmet ska fungera korrekt.

När man skriver en blocksats så förväntar sig Gandalf att du på efterföljande rad ökar indenteringsnivån, det vill säga lägger in ytterligare en tabb mer än förgående rad. När blocksatsen ska avslutas så förväntas en tabb mindre än förgående rad, alltså i samma nivå som starten för den sats som ska avslutas. Här kommer ett exempel som gör det hela tydligare:

```
if (x > y)
  if (y > z)
    print "y is greater than z"
  print "x is greater than y"
```

Observera också att Gandalf inte tillåter mellanslag (spaces) som indentering. Hårda tabbar måste användas.

2.3 Datatyper

Under detta avsnitt presenteras de datatyper som är implementerade i Gandalf.

2.3.1 Heltal

För att göra olika beräkningar med tal så skriver man den typ utav beräkning man vill göra med hjälp av någon utav de standardiserade operatorerna som de flesta programmeringsspråk tillämpar. Om du ändå känner dig osäker på dessa så finns de operatorerna som Gandalf tillämpar i del 2.4 av denna rapport.

```
a = 2 + 4 * 5
```

2.3.2 Logiska värden

Att arbeta med booleska värden är lika lätt som att arbeta med tal i Gandalf. De booleska värden som finns är `true` och `false`. Dessa ska alltid skrivas med små bokstäver. Även dessa värden har operatorer som finns uttryckta i del 2.4 av denna rapport.

```
a = true
b = false
print a && b
```

2.3.3 Strängar

Strängar är en sekvens av tecken som kan vara användbara på många sätt. Vi har tidigare sett exempel på hur man gör en utskrift under 2.1.1. I detta exempel använde vi oss utav en sträng. Strängar definieras genom att man sätter ett "-tecken följt av den text man vill ha i sin sträng och avslutar med ytterligare ett "-tecken. För att skapa en sträng skriver du följande:

```
a = "Detta är innehållet utav en sträng"
```

2.3.4 Listor

Listor är en konstruktion som klassas som en container. Det innebär att det är en konstruktion för att hålla flera olika värden. I språket Gandalf används listor främst för iteration i for-loopar.

Det finns huvudsakligen två sätt att skapa en lista. Antingen anger du ett start- och slutvärde och en lista kommer att deklareras med de värden som finns där emellan (inklusive start- och slutvärden). Denna metod fungerar bara för heltal men är väldigt smidig vid for-loopar där man vill att en viss sak ska göras ett specifikt antal gånger.

```
a = {1..100}
```

Du kan även skapa listor där du själv bestämmer vilket innehåll dessa ska ha. Det gör du på följande sätt:

```
a = {"Frodo", "Samwise", "Boromir", "Pippin", 2, 5, 9}
```

Om du vill använda saker i en lista kan du göra detta genom att använda en notation som gör att du kommer åt ett element med ett visst index. Det första elementet har index 0, nästa har 1 osv. Så här gör du för att komma åt strängen "Boromir" i listan som deklarerades i ovanstående exempel:

```
b = a{2}
```

2.4 Operatorer

I språket Gandalf finns ett antal olika operatorer som kan appliceras dels på artimetiska uttryck i form utav siffror, dels på sanningsvärden. Här följer en lista på dessa.

Tabell 1

Operator	Beskrivning
x || y	Disjunktion - or - eller
x && y	Konjunktion - and - och
! x	Negation - not - inte
x < y	Jämförelse - Less than - Mindre än
x > y	Jämförelse - Greater than - Större än
x <= y	Jämförelse - Less than or equal to - Mindre än eller lika med
x >= y	Jämförelse - Greater than or equal to -Större än eller lika med
x == y	Jämförelse - Equal - Lika med

Operator	Beskrivning
x != y	Jämförelse - Not equal - Inte lika med
x + y	Addition
x - y	Subtraktion
x * y	Multiplikation
x / y	Division
x << y	Används vid iteration i for-loopen. x är iterationsvariabeln för alla element i y

2.5 Villkorssatsen

Villkorssatsen if fungerar på samma sätt i Gandalf som i många andra språk. If-satsen är en grundläggande sats som styr hur programmet kommer att fortsätta exekveras beroende på om ett villkor är uppfyllt eller inte. Här följer ett exempel på hur detta ser ut:

```
if (gandlaf_is_late)
  print "A wizard is never late, Frodo Baggins."
elseif (!gandlaf_is_late)
  print "Nor is he early."
else
  print "He arrives precisely when he means to."
```

2.6 Repetitionssatsen

Gandalf implementerar två olika typer utav repetitionssatser. Dessa är while- och for-loopen.

2.6.1 While-loopen

En while-loop exekverar ett kodblock upprepade gånger så länge ett villkor är uppfyllt. Det betyder att man i denna loop någon gång måste göra villkoret falskt för att programmet ska fortsätta och inte fastna. While-loopen är en repetitionssats som finns i väldigt många språk och även denna sats fungerar på ett liknande sätt i Gandalf.

```
i = 10
while (i > 0)
  print "They taking the hobbits to Isengard!"
  i = i - 1
```

2.6.2 For-loopen

For-loopen är en loop som ofta används för att iterera över listor och detta är något vi har valt att ta vara på. I många andra språk finns en konstruktion för att först initiera ett värde, sedan deklarera ett villkor och slutligen tala om vad som ska hända med det initierade värdet för varje iteration. I Gandalf fungerar denna loop något annorlunda då Gandalf kräver att en iteration görs på en lista för att for-loopen ska kunna användas.

```
legolas_kill_count = {1..100}
for (i << legolas_kill_count)
  print i
```

Översta raden i kodexemplet skapar en lista med element från 1 till och med 100. För varje varv i loopen sätts variabeln `i` till det värde av det element som står på tur. Alltså – vid första iterationen har `i` värdet 1, i nästa iteration har den värdet 2 och så vidare. Detta avslutas sedan när `i` når listans slut med värdet 100 i vårt exempel.

2.7 Funktioner

2.7.1 Anrop

För att introducera funktioner behövs en förklaring på hur funktionsanropen är konstruerade. Många språk använder sig utav funktioner i form utav ett namn på funktionen följt utav en lista med inparametrar. Eftersom att det ofta är väldigt otydligt vad dessa inparametrar är och vilken variabel de tillhör har Gandalf inspirerats utav Objective-C:s sätt att lösa detta. Här kommer ett exempel på hur ett funktionsanrop brukar se ut i många andra språk:

```
calcVolumeForCube(10,15,20)
```

Med många inparametrar blir det otydligt vilket värde som är vilket. I Gandalf ser samma funktionsanrop ut så här:

```
[self, calc_volume_for_cube_with_height:10, width:15, depth:20]
```

Här råder det inte längre några tvivel vilket värde som tillhör vilken variabel. Nyckelordet self i början av funktionsanropet påvisar att funktionen är deklarerad i huvudprogrammet snarare än i en klass.

2.7.2 Definition

En funktion definieras på följande enkla sätt i Gandalf:

```
calc_volume_for_cube_with_height:height, width:width, depth:depth
  return height*width*depth
```

Funktionsnamnet är alltså uppdelat på så sätt att parametrar och funktionsnamn blandas på samma deklarationsrad. Namnet följer fram tills ‘:’-tecknet där vi väljer att ta emot en parameter. Detta värdet sparas till en lokal funktionsvariabel med namnet `height`. Ett kommatecken sätts för att separera och därefter fortsätter namnet tills nästa ‘:’-tecken. Där tas ytterligare en parameter emot osv.

2.7.3 Return-satsen

Return-satsen är en sats som används i en funktion för att få funktionen att returnera ett värde, t.ex. kan det vara så att du deklarerat en funktion för att beräkna något. För att sedan returnera detta värde använder du return-satsen såhär:

```
multiply:x with:y
  a = x * y
  return a
a = [self, multiply:3 with:3]
```

Efter ovanstående kodblock är kört har a värdet 9.

2.8 Klasser

Klasser är en konstruktion som används för att samla relaterade värden och funktioner.

2.8.1 Instansvariabler

Klasser i Gandalf har instansvariabler, men inte klassvariabler. Instansvariabler kallas de värden som sätts för den specifika instansen av klassen. Exempelvis skulle det kunna finnas en klass "Bil" som har instansvariabler som talar om vilken färg den specifika bilen har. Instansvariabler skapas på följande sätt med ett ‘@’-tecken innan variabelnamnet:

```
@num = 10
```

Du kan inte skapa instansvariabler i något annat än i en klass. Att skapa instansvariabler i huvudprogrammet vore inte logiskt då huvudprogrammet bara kommer finnas i en uppsättning.

2.8.2 Använda klasser

För att skapa en instans utav en viss klass kallar man på en så kallad konstruktor/initializer. Dessa kallas för init-funktioner i Gandalf. Att returnera värden ifrån init-funktioner är inte möjligt då init-funktioner alltid returnerar instanser utav den givna klassen. Funktionen init måste inte heta just detta, men måste börja på ordet init för att klassas som en initierare/konstruktor. Ett giltigt namn skulle t.ex. kunna vara init_with_number.

```
class Foo
  init
    @num = 0
    print "You ran the init-function!"
  init_with_number:number
    @num = number
    print "You assigned an instance variable @num"
  get_number
    return @num
a = [Foo, init]
b = [a, get_number]
```

Lägg märke till att det första funktionsanropet inte kallar på init på self då denna funktionen inte är deklarerad i huvudprogrammet utan i klassen Foo.

Efter att ha skapat en instans kan man sedan använda de funktioner som är deklarerade som inte är init-funktioner. Dessa kallas på den specifika instansen som exemplifieras ovan i det undre funktionsanropet.

2.9 Räckvidd

Viktigt att tänka på när man använder variabler är deras räckvidd. Beroende på var man deklarerar en variabel så har den en viss räckvidd som gör att man inte kommer åt den överallt. När man definierar en klass eller funktion så kommer man inte åt de variabler som deklarerats utanför. Följande fungerar alltså inte:

```
a = 5
b = 2
multiply
   return a*b
```

Däremot kommer man åt variabler i andra konstruktioner, t.ex. i while-loopar vilket man kan se i 2.6.1.

3. Systemdokumentation

3.1 Systemet

3.1.1 Lexikalisk analys

Vi använder oss utav RDparsern för att göra både en lexikalisk analys och för att parsa den skrivna Gandalf-koden.

RDparsern börjar med den lexikaliska analysen, där tokens skapas. En token är en sekvens utav tecken som är kategoriserad baserat på speciella regler, ofta i form utav reguljära uttryck. En programkod består i regel utav många olika tokens som sedan används vid parsningen.

Först plockar RDparsern ut och gör tokens utav radbrytningar och tabbar. Samtidigt som detta händer så har vi modifierat RDparsern så att den håller reda på indenteringen genom att räkna tabbarna för att sedan ta bort dessa. Istället för att använda alla tabbar som tokens så låter vi RDparsern generera tokens då indenteringsnivån har förändrats. Vi kan därför genom vår grammatik kräva att indenteringsnivån ökar eller minskar på givna ställen.

RDparsern går sedan igenom och gör tokens av nyckelorden class, return och print eftersom vi kräver mellanslag efter dessa nyckelorden. Därefter matchas sedan strängar, alltså allt innanför citattecken, eftersom vi vill kunna använda all text, även mellanslag, i dessa. Efter denna process är vi klara med alla mellanslag och väljer därför att inte behålla de återstående. Därefter går RDparsern vidare till att matcha de återstående nyckelorden nämligen; `true`, `false`, `if`, `for`, `else`, `while`, `&&` och `||`. Den gör sedan tokens utav alla återstående tecken var för sig.

3.1.2 Parsning

Parsern kontrollerar den skrivna gandalf-koden baserat på de regler som finns i grammatiken. I grammatiken uttrycks vilka sammansättningar utav tokens som är korrekta för att språket ska kunna tolkas på lämpligt sätt. Efter den lexikaliska analysen så börjar RDparsern parsningen, vilken använder sig utav följande grammatik för Gandalf-språket:

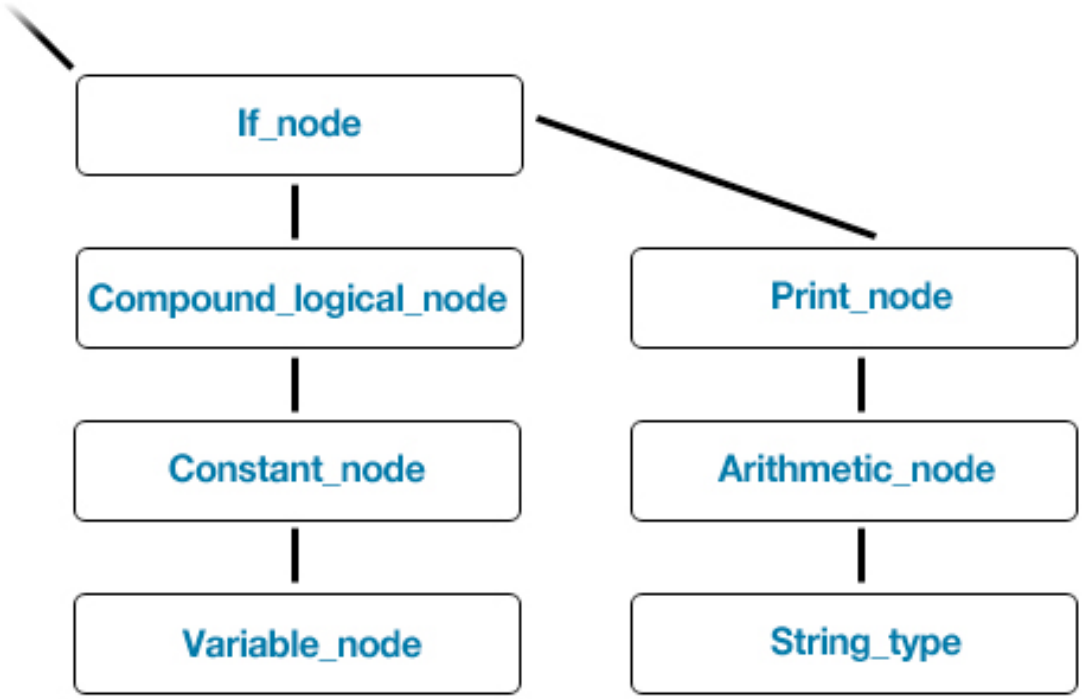
```
program ::= statement_list
statement_list ::= statement [statement_list]
statement ::= (complex_statement | simple_statement + NEWLINE)
complex_statement ::= (class_def | function_def | while_stmt | if_stmt |
for_stmt)
simple_statement ::= (return_stmt | function_call | print_stmt | assignment_stmt
| expr)
## Complex statements
class_stmt ::= "class" + indentifier + NEWLINE + INDENT + class_body + DEDENT
function_def ::= (function_part | function_name) + suite
for_stmt ::= "for" + "(" + identifier + "<<" + expr + ")" + suite
while_stmt ::= "while" + "(" + expr + ")" + suite
if_stmt ::= "if" + "(" + expr + ")" + suite + ["else" + (suite | if_stmt)]
## Simple statements
return_stmt ::= "return" + expr
function_call ::= "[" + identifier + "," + (function_call_part | function_name)
+ "]"
print_stmt ::= "print" + expr
assignment_stmt ::= identifier + "=" + (expr | function_call)
expr ::= or_test | function_call
class_body ::= function_def + [class_body]
function_part ::= function_name + ":" + identifier + ["," + function_part]
function_call_part ::= function_name + ":" + expr + ["," + function_call_part]
suite ::= NEWLINE + INDENT + statement_list + DEINDENT
function_name ::= identifier
logical_operator ::= ("<" | ">" | "=") | ("<" | ">" | "=" | "!") + "="
or_test ::= and_test | or_test + "||" + and_test
and_test ::= not_test | :and_test + "&&" + not_test
not_test ::= comparison | "!" + not_test
comparison ::= a_expr | a_expr + logical_operator + a_expr
a_expr ::= m_expr | a_expr + ("+" | "-") + m_expr
m_expr ::= u_expr | m_expr + ("*" | "/" | "%") + u_expr
u_expr ::= atom | "-" + u_expr
atom ::= "(" + comparison + ")" | boolean | number | identifier | "@" +
identifier | list | list_accessor
identifier ::= identifier_chars + [identifier_part]
identifier_part ::= identifier_chars + [identifier_part] | digit +
[identifier_part]
identifier_chars ::= A..Z | a..z | _
boolean ::= "true" | "false"
```

```
number ::= float | integer
string ::= '"' + string_body + '"'
list ::= "{" + list_body + "}" | "{" + expr + ".." + expr + "}"
list_body ::= expr + "," + list_body | expr
list_accessor ::= identifier + "{" + number + "}"
string_body ::= [^"]
float ::= integer + "." + integer
integer ::= digit + [integer]
digit ::= 0..9
```

3.1.3 Trädstruktur

När RDparsern parsat och matchat en konstruktion som t.ex. ett hetal, så skapas en nod för att hantera värdet. Vårt exempel med ett heltal blir en `Constant_node`. För alla konstruktioner i språket finns en matchande nod som har hand om den specifika konstruktionen. Alla noder är implementerade som en uppsättning av klasser. Alla dessa har en funktion som heter `eval` som används för att evaluera noden. Vi har flera noder som innehåller undernoder av olika slag och på så sätt bygger vi upp en trädstruktur utav alla skapade noder. Eftersom alla noder har en evalueringsfunktion, så kan man lätt köra hela den här trädstrukturen när den är färdigbyggd. Program-noden, som är den översta i hela trädstrukturen, håller under sig hela programmet i form utav noder av olika slag. Genom att köra evalueringsfunktionen på den översta noden kommer den i sin tur att kalla sina underliggande noders evalueringsfunktion tills dess att man nått botten utav det uppbyggda trädet då det uppnådda resultatet returneras.

Här följer ett exempel på hur trädstrukturen kan se ut:



3.1.4 Variabel- och scopehantering

Scope, eller räckvidd, är något man har för variabler för att tala om vilken räckvidd den har. Beroende på var en variabel är deklarerad har den olika lång räckvidd. T.ex. ska en variabel som är deklarerad i en villkorssats inte finnas när man avslutar satsen.

Variabelhanteringen är implementerad med hjälp utav en global lista. Varje position i listan är ett scope. På index noll finns det scope som programmet börjar med och som också har störst räckvidd. På index ett finns nästa scope osv. Ett nytt scope skapas varje gång man träder in i en blocksats, som sedan försvinner när blocksatsen avslutas.

Varje scope representeras av en hashtabell som innehåller scopets variabler och funktioner. Ytterligare en global variabel håller reda på vilket scope vi befinner oss i under exekveringen.

För att hantera att man ibland inte vill komma åt variabler som är deklarerade på en lägre nivå än ett visst scope har vi implementerat något som vi kallar för basscope. Ett exempel på när detta är användbart är t.ex. när man kör en funktion. I funktioner vill man inte komma åt variabler som är deklarerade utanför funktionen själv och som inte skickats som parameter. För att göra detta har vi skapat en stack som håller reda på de olika basscope som finns vid ett givet tillfälle. När vi träder in i ett basscope så lägger vi till scopets nivå till stacken och denna tas sedan bort när vi kliver ut ur basscopet.

När man letar efter en variabel så kommer man börja med att kolla i den hashtabell som ligger i det basscope som ligger överst på stacken. Om ingenting hittas i detta scope så stegar man uppåt tills dess att man hittar variabeln eller inte har några övre scope att leta i. Hittas inte detta så finns ingen variabel deklarerad för detta namn.

3.2 Kodstandard

Vi har valt att använda snake case som kodstandard, eftersom vi båda ansåg att det bidrar till mer läsbar kod. Klassnamn bör vara de enda namnen som börjar med stor bokstav. Denna standard har vi använt under implementationen utav språket, och är också den standard vi anser att man bör använda i språket Gandalf.

Indentering är ett krav i språket Gandalf och man kan inte dela upp en rad kod hur man vill. Detta valde vi att göra för att vi inte tyckte att användaren skulle kunna komma undan med att skriva ostrukturerad kod.

3.3 Installation

Denna installationsguide utgår från att man använder Ubuntu. Följande installationsmoment måste utföras oberoende av vilket operativsystem man använder. Tillvägagångssättet för de olika installationerna skiljer sig dock mellan olika system. Den här guiden kommer bara förklara hur man går tillväga under operativsystemet Ubuntu.

3.3.1 Gandalf

Filerna som behövs för att använda Gandalf går att hitta på [länk] och är paketerade i en tarball-fil vid namn gandalf-1.0.tar. Paketet innehåller parser.rb, rdparser.rb, library.rb,

gandalf.rb och tree.rb. Den innehåller även dokumentationen och grammatiken, vilka ligger i mappen Doc. De filer som ligger i mappen Doc är inte obligatoriska för att kunna köra språket Gandalf. Filstrukturen för paketet ser ut som följer:

```
/
     Doc/
          Example code/
               *.gdf
          grammar.txt
          Rapport.pdf
     gandalf.rb
     library.rb
     parser.rb
     rdparser.rb
     tree.rb
```

För att packa upp en tarball skriv följande kommandon i terminalen:

```
cd /sökväg/till/katalog/där/gandalf-1.0.tar/ligger
tar zxvf gandalf-1.0.tar
```

3.3.2 Ruby

Programmet är utvecklat och testat för Ruby 1.8.5 och man bör därför ha detta installerat. För att göra detta skriver man "`sudo apt-get install ruby`" i terminalen.

3.3.3 Interpretatorn

För att starta interpretatorn går du till den katalog där Gandalf.rb finns. Sedan skriver du:

```
ruby gandalf.rb
```

Vill du hellre köra en fil. Skriv följande:

```
ruby gandlaf.rb file.gdf
```

4. Erfarenheter och reflektion

Projektet var både lättare och svårare än vi trott. Själva grunden, att göra ett språk med RDparsern som verktyg, var lättare än vi trodde. Däremot var det svårare att göra några utav de konstruktioner som vi hade valt att implementera i vårt språk.

4.1 Indentering

Indentering är ett bra exempel på ett moment som tog mycket längre att implementera än vad vi hade tänkt oss. Det som gjorde att vi upplevde det som svårt var att vi inte visste vart vi skulle börja skriva koden för att hantera indentering. Vi försökte hitta en lösning som gjorde att vi inte behövde modifiera RDparsern. Detta var dock väldigt besvärligt då så fort man hade matchat en tabb blev tvungen att returnera något. Detta skapade problem eftersom vi ville att den bara skulle returnera tokens då indenteringsnivån hade förändrats.

För att lösa detta fick vi börja med att studera koden i RDparsern för att lista ut vart den här koden borde skrivas istället. Resultatet blev att vi skrev några rader kod i RDparserns tokenizer-funktion. Om vi stötte på en radbrytning så kontrollerade vi om nästa token var tabbar. Var så fallet så räknade vi dessa och jämförde med tidigare antal. Om antalet tabbar skiljer sig ifrån indenteringsnivån så returnerar RDparsern en INDENT-token eller en DEDENT-token beroende på om nivån ökat eller minskat.

4.2 Klasser

En annan konstruktion som också var svårare än vad vi förväntade oss var att göra klasser. Bara grunden, att göra en klass, var inte så svårt. Att sedan göra det möjligt att skapa instanser och instansvariabler var mycket svårare, och krävde mycket tid och kod. Vi behövde göra väldigt många kontroller i vår Function_call_node för att se om vi kallade funktionen på en klass, om vi kallade på en init-funktion, om init-funktion fanns deklarerad, om det fanns en funktion som bara börjar på init (eftersom vi bestämt att det också borde fungera) eller om det är en instans vi kallade funktioner på. Detta gjorde att vi fick ganska rörig och oöverskådlig kod i denna nod, men tyvärr kunde vi inte komma runt det problemet.

4.3 Booleska värden

När vi implementerade hanteringen utav booleska värden blev vi tvungna att göra modifieringar i RDparsern. När matchnings resultatet var det booleska värdet false från Gandalf så returnerades nil eftersom att match_result var false. Det värde vi egentligen ville ha var false. För att lösa detta bytte vi följande:

```
return nil unless match_result
```

Mot:

```
return nil if match_result == nil
```

4.3 Mål vi hade och uppnådde

Vi hade som mål att göra ett interpreterat språk med en tilltalande, lättläslig och samtidigt något unik syntax. Detta anser vi båda att vi har lyckats uppnå med vårt språk.

Vi hade idéen om att göra ett både starkt- och statiskt typat språk med en mängd olika datatyper, men så långt hann vi tyvärr aldrig. Detta är dock något som vi inte är så besvikna över då det kändes relativt naturligt att låta detta språk vara dynamiskt typat. Vi hade även som plan att implementera arv till våra klasser. Detta hann vi tyvärr inte heller med, men hade vi haft mer tid hade vi gärna gjort det.

Som helhet är vi båda nöjda med det arbete vi har lagt ner i detta projekt och känner att vi uppnått det resultat vi förväntade oss, även om vi inte hann med alla våra mål. Vi känner att vi lärt oss väldigt mycket om hur datorspråk är uppbyggda och att vi fått en betydligt mer grundlig förståelse för språkuppbyggnad. Att jobba med en trädstruktur på detta sättet kändes också riktigt kul.

5. Bilagor

Gandalf.rb

```
require "parser"
@wizard = Gandalf.new

def read_manual_input(code = "")
     if code == ""
          print "Gdf> "
     else
          print ".... "
     end
     line = gets
     if(line == "\n" && code != "")
          code += line
          puts "=> #{@wizard.parser.parse(code)}"
          read_manual_input
     elsif(!["quit","exit"].include?(line.chomp))
          code += line if line != "\n"
          read_manual_input(code)
     end
end
def read_file_input(code = "")
     input = gets
     if (input)
          read_file_input(code += input)
     else
          puts "=> #{@wizard.parser.parse(code)}"
     end
end
if ARGV.length > 0
     read_file_input
else
     read_manual_input
end
```

Library.rb

```
class String_type
     attr_accessor :value
     def initialize(data)
          @value = data
     end
     def eval
          return @value
     end
end

class List_type
     attr_accessor :list_values, :range_value
     def initialize(first_value)
          @list_values = [first_value]
     end
     def eval
          if range_value
               @list_values = (@list_values[0]..@range_value).to_a
          end
          return @list_values
     end
end
```

Parser.rb

```
# -*- coding: utf-8 -*-
require "rdparser"
require "tree"
require "library"

class Gandalf
  attr_accessor :parser
  def initialize
    @code = ""
    @parser = Parser.new("Gandalf") do
      token(/\n+/){ |m| "\n" }
      token(/\t+/)
      token(/class\s/){ |m| "class" }
      token(/return\s/){ |m| "return" }
      token(/print\s/){ |m| "print" }
      token(/"[^"]*"/){|m| m}
      token(/\s+/)

      token(/true/){ |m| m }
      token(/false/){ |m| m }
      token(/if/){ |m| m }
      token(/for/){|m| m}
      token(/else/){ |m| m }
      token(/while/){ |m| m }
      token(/\&\&/){ |m| m }
      token(/\|\|/){ |m| m }
      token(/./){ |m| m }

      start :program do
        match(:statement_list){ |a| a.eval }
      end

      rule :suite do
        match("\n", "→", :statement_list, "←"){ |_,_,a,_| a }
      end

      rule :statement_list do #this is our statement+
        match(:statement,:statement_list){ |a,b|
Compound_statement_list_node.new(a,b) }
```

```
        match(:statement)
      end

      rule :statement do
        match(:complex_stmt)
        match(:simple_stmt, "\n"){ |a,_| a }
      end

      rule :complex_stmt do
        match(:class_def)
        match(:function_def)
        match(:for_stmt)
        match(:while_stmt)
        match(:if_stmt)
      end

      rule :simple_stmt do
        match(:return_stmt)
        match(:function_call)
        match(:print_stmt)
        match(:assignment_stmt)
        match(:expr)
      end

      rule :print_stmt do
        match("print",:expr){ |_,a| Print_stmt_node.new(a) }
      end

      rule :class_def do
        match("class", :identifier,"\n", "→", :class_body, "←"){ |_,a,_,_,b,_|
Class_node.new(a,b) }
      end

      rule :class_body do
        match(:function_def, :class_body) { |a,b|
Compound_statement_list_node.new(a,b) }
        match(:function_def)
      end

      rule :function_def do
```

```
        match(:function_part, :suite){ |a,b| a.function_body = b
          a
        }
        match(:function_name,:suite){ |a,b| c =Function_node.new(a)
          c.function_body = b
          c
        }
      end

      rule :function_part do
        match(:function_name, ":", :identifier, "," , :function_part){ |
a,_,b,_,c|
          c.function_name = a + "_" + c.function_name
          c.variable_names.insert(0,b)
          c
        }
        match(:function_name, ":", :identifier){ |a,_,b| Function_node.new
(a,b) }
      end

      rule :function_name do
        match(:identifier)
      end

      rule :function_call do
        match("[", :identifier, ",", :function_call_part,"]"){ |_,a,_,b,_|
          b.type = a
             b
           }
        match("[", :identifier, ",", :function_name,"]"){ |_,a,_,b,_|
          c = Function_call_node.new(b)
          c.type=a
          c
        }
      end

      rule :function_call_part do
        match(:function_name, ":", :expr, ",", :function_call_part){ |a,_,b,_,c|
          c.name = a + "_" +c.name
          c.variable_values.insert(0,b)
```

```
          c
        }
        match(:function_name, ":", :expr) { |a,_,b| Function_call_node.new
(a,b) }
      end

      rule :return_stmt do
        match("return", :expr){ |_,a| Return_statement_node.new(a) }
      end

      rule :while_stmt do
        match("while", "(", :expr, ")", :suite){ |_,_,a,_,b,| While_node.new
(a,b) }
      end

              rule :for_stmt do
                   match("for", "(", :identifier, "<","<", :expr, ")", :suite){|
_,_,a,_,_,b,_,c|For_node.new(a,b,c)}
              end

      rule :if_stmt do
        match("if","(", :expr, ")", :suite, "else", :suite){ |_,_,a,_,b,_,c|
If_else_node.new(a,b,c) }
        match("if","(", :expr, ")", :suite, "else", :if_stmt){ |_,_,a,_,b,_,c|
If_else_node.new(a,b,c) }
        match("if","(", :expr, ")", :suite){ |_,_,a,_,b| If_node.new(a,b) }
      end

      rule :assignment_stmt do
        match(:identifier, "=", :expr){ |a,_,b| Assignment_node.new
((Variable_node.new(a)),b) }
        match(:identifier, "=", :function_call){ |a,_,b| Assignment_node.new
((Variable_node.new(a)),b)}
        match("@", :identifier, "=", :expr){|_,a,_,b|
Instance_assignment_node.new((Instance_variable_node.new(a)),b)}
        match("@", :identifier, "=", :function_call){|_,a,_,b|
Instance_assignment_node.new((Instance_variable_node.new(a)),b)}
      end

      rule :expr do
```

```
        match(:or_test)
                    match(:function_call)
      end

      rule :or_test do
        match(:and_test)
        match(:or_test, "||", :and_test){ |a,b,c| Compound_logical_node.new
(a,b,c) }
      end

      rule :and_test do
        match(:not_test)
        match(:and_test, "&&", :not_test){ |a,b,c| Compound_logical_node.new
(a,b,c) }
      end

      rule :not_test do
        match(:comparison)
        match("!", :not_test){ |_,a| Not_node.new(a) }
      end

      rule :comparison do
        match(:a_expr, :logical_operator, :a_expr){ |a,b,c|
Compound_logical_node.new(a,b,c) }
        match(:a_expr){ |a| Arithmetic_node.new(a) }
      end

      rule :logical_operator do
        match(/<|>|=|!/,/=/){ |a,b| a+b }
        match(/<|>|=/)
      end

      rule :a_expr do
        match(:m_expr)
        match(:a_expr, "+", :m_expr) { |a,b,c| Compound_arithmetic_node.new
(a,b,c) }
        match(:a_expr, "-", :m_expr) { |a,b,c| Compound_arithmetic_node.new
(a,b,c) }
      end
```

```
      rule :m_expr do
        match(:u_expr)
        match(:m_expr, "*", :u_expr){ |a,b,c| Compound_arithmetic_node.new
(a,b,c) }
        match(:m_expr, "/", :u_expr){ |a,b,c| Compound_arithmetic_node.new
(a,b,c) }
        match(:m_expr, "%", :u_expr){ |a,b,c| Compound_arithmetic_node.new
(a,b,c) }
      end

      rule :u_expr do
        match(:atom)
        match("-", :u_expr){ |_,a| a * -1 }
      end

      rule :atom do
        match("(", :comparison, ")"){ |_,a,_| a }
        match(:boolean)
        match(:number)
                  match(:list_access)
                  match(:string)
        match(:identifier){ |a| Variable_node.new(a) }
                  match("@", :identifier){|_,a| Instance_variable_node.new(a)}
                  match(:list)
      end

      rule :identifier do
        match(:identifier_chars, :identifier_part){ |a,b| a+b }
        match(:identifier_chars)
      end

      rule :identifier_part do
        match(:identifier_chars, :identifier_part) { |a,b| a+b }
        match(:digit, :identifier_part) { |a,b| a+b }
        match(:identifier_chars)
        match(:digit)
      end

              rule :list_access do
```

```
                    match(:identifier, "{", :number, "}"){|a,_,b,_|
List_access_node.new(b, Variable_node.new(a))}
               end

               rule :list do
                    match("{",:list_body,"}"){|_,a,_|a}
                    match("{", :expr, ".", ".", :expr, "}"){|_,a,_,_,b,_|c =
List_type.new(a.eval)
                    c.range_value = b.eval
                    c
                    }
               end

               rule :list_body do
                    match(:expr, ",", :list_body){|a,_,b|b.list_values.insert
(0,a.eval)
                    b}

                    match(:expr) {|a|List_type.new(a.eval)}
               end

               rule :string do
                    match(/"[^"]*"/){|a|String_type.new(a)}
               end

               rule :string_body do
                    match(/[^"]/, :string_body){|a,b|a+b}
                    match(/[^"]/){|a|a}
               end

      rule :boolean do
        match("true") { |a| Constant_node.new(true) }
        match("false") { |a| Constant_node.new(false) }
      end

      rule :number do
        match(:float){ |a| Constant_node.new(a.to_f) }
        match(:integer){ |a| Constant_node.new(a.to_i) }
      end
```

```
      rule :float do
        match(:integer,".",:integer){ |a,_,b| a+"."+b }
      end

      rule :integer do
        match(:digit, :integer) { |a,b| a+b }
        match(:digit){ |a| a }
      end

      rule :identifier_chars do
        match(/[A-Za-z_]/)
      end

      rule :digit do
        match(/[0-9]/)
      end
    end
  end
end
```

Tree.rb

```
@@identifiers =[{}] #This is used to handle our scope, the lists index indicates what scope to access
@@classes = {} #We wanted to be able to access classes everywhere, so we created a new hashtable for handeling classes
@@scope = 0 #Indicates which index we should use in @@identifiers
@@base_scope = [0] #Is used when a new function or class is created to hinder access to variables outside of them
@@instance_stack = [] #Is used to associate instance variables with the right instance
@@instance_variables = {} #Is used for handeling instance variables

def look_up(variable)
  i = @@scope
  while(i>=@@base_scope.last)
    if @@identifiers[i][variable] != nil
      return @@identifiers[i][variable]
    end
    i -= 1
```

```
  end
  if variable.match(/^0/)
    raise(NameError, "Function '#{variable}' does not exist. Your code shall not
pass!")
  else
    raise(NameError, "Variable '#{variable}' does not exist. Your code shall not
pass!")
  end
end

def new_scope
  @@scope +=1
  @@identifiers << {}
end

def new_base_scope(move_functions_to_new_base_scope = false)
  #This ensures that we cannot access variables that were created outside of
this scope.
  #For example, functions use this.

  functions = {}
  if (move_functions_to_new_base_scope)
    # Enter this loop if functions should be moved into the new base scope
    i = @@scope
    while(i>=@@base_scope.last)
      @@identifiers[i].each_pair do | key, value |
        if (key.match(/^0_/))
          functions[key] = value
        end
      end
      i -= 1
    end
  end

  new_scope()
  @@base_scope << @@scope

  # Add every function that should be moved to new scope
  functions.each_pair do | key, value |
    @@identifiers[@@scope][key] = value
```

```
  end
end

def close_scope
  @@identifiers.pop
  @@scope-=1
  if @@scope < 0
    raise("Scope is now less then 0, check your code Mr Frodo")
  end
end

def close_base_scope
  close_scope
  @@base_scope.pop
end

class Compound_statement_list_node
  attr_accessor :statement1, :statement2
  def initialize(stmt1, stmt2) #stmt2 can either be a statement or another
compound_statement_list_node
    @statement1 = stmt1
    @statement2 = stmt2
  end

  def eval
    return_val = @statement1.eval
    if @statement1.class != Return_statement_node #If statement1 is a return
statement we do not run the rest of the code
      @statement2.eval
    else
      return return_val
    end
  end

     def find_functions
    if @statement2.respond_to? :find_functions
      # statement2 seems to be another compound statement list
      function_list = @statement2.find_functions()
      if @statement1.function_name
        function_list << @statement1
```

```
      end
      return function_list
    else
      # Quite messy, but this is the collecting part of all functions
      function_list = []
      if @statement1.function_name
        function_list << @statement1
      end
      if @statement2.function_name
        function_list << @statement2
      end
      return function_list
    end
  end
end

class Function_node
  attr_accessor :function_name, :variable_names, :function_body
  def initialize(name_part, variable = nil)
    @function_name = name_part
    @variable_names = []
    @variable_names << variable if variable
    #@function_body = function_body
  end
  def eval
    @@identifiers[@@scope]["0_#{@function_name}"] = self
  end
end

class Function_call_node
  attr_accessor :name, :variable_values, :type
  def initialize(name_part, value= nil)
    @name = name_part
    @type = "self"
    @variable_values = []
    @variable_values << value if value != nil
  end

    def instance_name(name) #checks which instance(if any) we are currently
using
```

```
          if @@classes[@type]
               @is_instance = true
               @@instance_stack << name
          elsif @type != "self"
               @is_instance = true
               @@instance_stack << @type
          end
     end

  def run_function(function)
          eval_values = []
          (0..@variable_values.length-1).each do |j|
               eval_values[j] = @variable_values[j].eval # we evaluate our values
before we enter the new scope
          end

    new_base_scope(true)
    (0..function.variable_names.length-1).each do |i|
      @@identifiers[@@scope][function.variable_names[i]] = eval_values[i]
    end
    return_val = function.function_body.eval
    close_base_scope()

          if @is_instance
               @@instance_stack.pop
          end
    return return_val
  end

  def eval
    if @type == "self"
      # Function call on self
      function = look_up("0_#{@name}")
      return run_function(function)
    elsif @@classes[@type] and @name.match(/^init/)
      # Init call on class
      if @@classes[@type].class_body.respond_to? :find_functions
        function_list = @@classes[@type].class_body.find_functions
                for function in function_list
         if function.function_name == @name
```

```
            run_function(function)
            return @@classes[@type]
          end
        end
      elsif @@classes[@type].class_body.function_name.match(/^init/)
        #we only do this if there is only one function in the class
        run_function(@@classes[@type].class_body)
        return @@classes[@type]
      elsif @name == "init"
        return @@classes[@type]
      end
      raise(NameError, "Undefined function '#{@name}' for class '#{@type}'. You
fool of a Took!")
    else
      # Function call on instance
              instance_name(@type)
      class_node = look_up(@type)
      # Open new base scope so that functions defined outside class can't be
called from inside of class
      new_base_scope()
      class_node.class_body.eval
      function = look_up("0_#{@name}")
      return_val = run_function(function)
      close_base_scope()
      return return_val
    end
  end
end

class Return_statement_node
  attr_accessor :expression
  def initialize(expr)
    @expression = expr
  end
  def eval
    return @expression.eval
  end
end
```

```
class Class_node
  attr_accessor :name, :class_body
  def initialize(name, functions, inheritance=nil)
    @name = name
    @class_body = functions
    @inheritance = inheritance
  end
  def eval
    @@classes[@name] = self
  end
end

class If_else_node
  attr_accessor :condition, :statement1, :statement2
  def initialize(cond, stmt1, stmt2)
    @condition = cond
    @statement1 = stmt1
    @statement2 = stmt2
  end
  def eval
    new_scope()
    if @condition.eval
      return_value = @statement1.eval
    else
      return_value = @statement2.eval
    end
         close_scope()
    return return_value
  end
end

class Print_stmt_node
  attr_accessor :expression
  def initialize(expr)
    @expression = expr
  end
  def eval
    puts @expression.eval
  end
```

```
end

class If_node
  attr_accessor :condition, :statement
  def initialize(cond, stmt)
    @condition = cond
    @statement = stmt
  end
  def eval
    new_scope()
    if @condition.eval
      return_value =@statement.eval
      close_scope()
      return return_value
    end
    close_scope()
  end
end

class For_node
     def initialize(variable, list, loop_body)
          @iterator_var = variable
          @iteration_values = list
          @loop_body = loop_body
     end
     def eval
          new_scope()
          @iteration_values.eval.each do |i|
               Assignment_node.new(Variable_node.new(@iterator_var),
Constant_node.new(i)).eval
               @loop_body.eval
          end
          close_scope()
          nil
     end
end

class While_node
  attr_accessor :condition, :statement
  def initialize(cond, stmt)
```

```
    @condition = cond
    @statement = stmt
  end
  def eval
    new_scope()
    while @condition.eval do
      @statement.eval
    end
    close_scope()
    nil
  end
end

class Compound_node
  attr_accessor :operator, :operand1, :operand2
  def initialize(op1, op, op2)
    @operator = op
    @operand1 = op1
    @operand2 = op2
  end
  def eval
         if @operand1.class == String_type and @operand2.class == String_type
              return instance_eval("'#{@operand1.eval}' #{operator} '#
{@operand2.eval}'")
         end
    return instance_eval("#{@operand1.eval} #{operator} #{@operand2.eval}")
  end
end

class Not_node
  attr_accessor :operand
  def initialize(op)
    @operand = op
  end
  def eval
    return (not @operand.eval)
  end
end

class Compound_logical_node < Compound_node
```

```
end

class Assignment_node
  attr_accessor :identifier, :expr
  def initialize(id, expression)
    @identifier = id
    @expr = expression
  end
  def eval
         if @expr.respond_to? :instance_name
              @expr.instance_name(@identifier.name)
         end

    i = @@base_scope.last
    add_variable = true
    while(i<=@@scope)
      if @@identifiers[i][@identifier.name] != nil
        @@identifiers[i][@identifier.name] = @expr.eval
        add_variable = false
      end
      i+=1
    end
    if add_variable
      @@identifiers[@@scope][@identifier.name] = @expr.eval
    end
  end
end

class Instance_assignment_node
     attr_accessor :identifier, :expr
  def initialize(id, expression)
    @identifier = id
    @expr = expression
  end

     def eval
          if @@instance_stack.last
               if @@instance_variables[@@instance_stack.last]
                    @@instance_variables[@@instance_stack.last][@identifier.name]
= @expr.eval
```

```
                else
                    @@instance_variables[@@instance_stack.last] =
{@identifier.name => @expr.eval}
                end
            end
        end
end


class Constant_node
  attr_accessor :value
  def initialize(data)
    @value = data
  end
  def eval
    return @value
  end
end

class Variable_node
  attr_accessor :name
  def initialize(identifier)
    @name = identifier
  end
  def eval
    return look_up(@name)
  end
end

class Instance_variable_node
    attr_accessor :name
    def initialize(identifier)
        @name = identifier
    end
    def eval
        if @@instance_variables[@@instance_stack.last]
            return @@instance_variables[@@instance_stack.last][@name]
        else
            raise(NameError, "Instance variable '#{@name}' does not exist.
Your code shall not pass!")
```

```
          end
     end
end

class Compound_arithmetic_node < Compound_node
end

class Arithmetic_node
  attr_accessor :arithmetic_expr
  def initialize(arithmetic)
    @arithmetic_expr = arithmetic
  end
  def eval
    return @arithmetic_expr.eval
  end
end

class List_access_node
     attr_accessor :list_variable
     def initialize(index, variable)
          @index = index
          @list_variable = variable

     end
     def eval
          return @list_variable.eval[@index.eval]
     end
end
```

RDParser.rb

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

require 'logger'

class Rule
  # A rule is created through the rule method of the Parser class, like so:
  #   rule :term do
  #     match(:term, '*', :dice) {|a, _, b| a * b }
```

```
#     match(:term, '/', :dice) {|a, _, b| a / b }
#     match(:dice)
#   end

  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    # The name of the expressions this rule matches
    @logger=Logger.new(STDOUT)
    @logger.level = Logger::WARN
    @name = name
    # We need the parser to recursively parse sub-expressions
    # occurring within the pattern of the match objects associated
    # with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches, which in the first two cases
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #     match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.

  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
    if pattern[0] == @name
      pattern.shift
      @lrmatches << match
    else
      @matches << match
    end
  end

  def parse
    # Try non-left-recursive matches first, to avoid infinite recursion
    match_result = try_matches(@matches)
    return nil if match_result == nil
```

```
    #return false unless match_result #WE CHANGED RETURN false, USED TO BE
RETURN nil
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result unless result
      match_result = result
    end
  end


  private

  # Try out all matching patterns of this rule
  def try_matches(matches, pre_result = nil)
    match_result = nil
    # Begin at the current position in the input string of the parser
    start = @parser.pos
    matches.each do |match|
      # pre_result is a previously available result from evaluating expressions
      result = pre_result ? [pre_result] : []

      # We iterate through the parts of the pattern, which may be
      # [:expr,'*',:term]
      match.pattern.each_with_index do |token,index|

        # If this "token" is a compound term, add the result of
        # parsing it to the "result" array

        if @parser.rules[token]
          result << @parser.rules[token].parse
          unless result.last
            result = nil
            break
          end
          @logger.debug("Matched '#{@name} = #{match.pattern
[index..-1].inspect}'")
        else
          # Otherwise, we consume the token as part of applying this rule
          nt = @parser.expect(token)
          if nt
```

```
            result << nt
            if @lrmatches.include?(match.pattern) then
              pattern=[@name]+match.pattern
            else
              pattern=match.pattern
            end
            @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #
{pattern.inspect}'")
          else
            result = nil
            break
          end
        end
      end
      if result
        if match.block
          match_result = match.block.call(*result)
        else
          match_result = result[0]
        end
        @logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#
{@name}' and generated '#{match_result.inspect}'") unless match_result.nil?
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
      end
    end
    return match_result
  end
end

class Parser
  attr_accessor :pos
  attr_reader :rules,:string
  class ParseError < RuntimeError; end
  def initialize(language_name, &block)
    @logger=Logger.new(STDOUT)
    @logger.level = Logger::WARN
```

```
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name=language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @tab_stack = 0
    @check_tabs = false
    @tokens = []
    @string=string.clone
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
      raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |
tok|
        match = tok.pattern.match(string)
        # The regular expression of a token has matched the beginning of
'string'
        if match
          @logger.debug("Token #{match[0]} consumed")
          if match.to_s.match(/\n/) and !@check_tabs
            @check_tabs = true
          elsif @check_tabs
            if match.to_s.match(/\t+/)
              if match.to_s.length == @tab_stack + 1
                @tokens << "→"
                @tab_stack += 1
              end

              while(match.to_s.length < @tab_stack)
                @tokens << "←"
                @tab_stack -= 1
              end
            elsif @tab_stack > 0 #Goes back to tab level 0
              while(@tab_stack > 0)
                @tokens << "←"
```

```
                @tab_stack -= 1
              end
            end
            @check_tabs = false
          end
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
          true
        else
          # this token pattern did not match, try the next
          false
        end # if
      end # raise
    end # until
   #puts @tokens
    #print @tokens
  end

  def parse(string)
    # First, split the string according to the "token" instructions given to
Parser
    tokenize(string)
    # Now, @tokens contains all tokens that are to be parsed.

    while(@tab_stack > 0)
      @tokens << "←"
      @tab_stack -= 1
    end

    # These variables are used to match if the total number of tokens
    # are consumed by the parser
    @pos = 0
    @max_pos = 0
    @expected = []
    # Parse (and evaluate) the tokens received
    result = @start.parse
    # If there are unparsed extra tokens, signal error
```

```
    if @pos != @tokens.size
      raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found
'#{@tokens[@max_pos]}'"
    end
    return result
  end

  def next_token
    @pos += 1
    return @tokens[@pos - 1]
  end

  # Return the next token in the queue
  def expect(tok)
    t = next_token
    if @pos - 1 > @max_pos
      @max_pos = @pos - 1
      @expected = []
    end
    return t if tok === t
    @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
    return nil
  end

  def to_s
    "Parser for #{@language_name}"
  end

  private

  LexToken = Struct.new(:pattern, :block)

  def token(pattern, &block)
    @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
  end

  def start(name, &block)
    rule(name, &block)
    @start = @rules[name]
  end
```

```
  def rule(name,&block)
    @current_rule = Rule.new(name, self)
    @rules[name] = @current_rule
    instance_eval &block
    @current_rule = nil
  end

  def match(*pattern, &block)
    @current_rule.send(:match,*pattern,&block)
  end
end
```