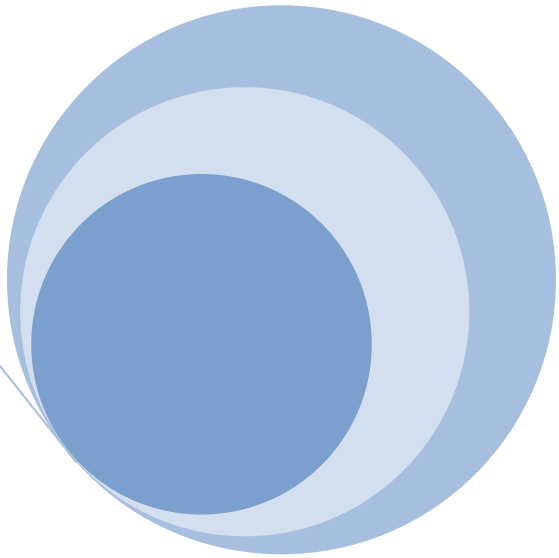
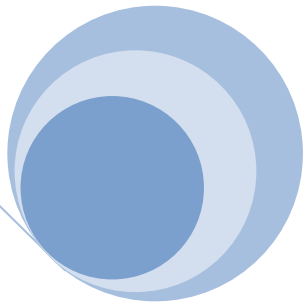




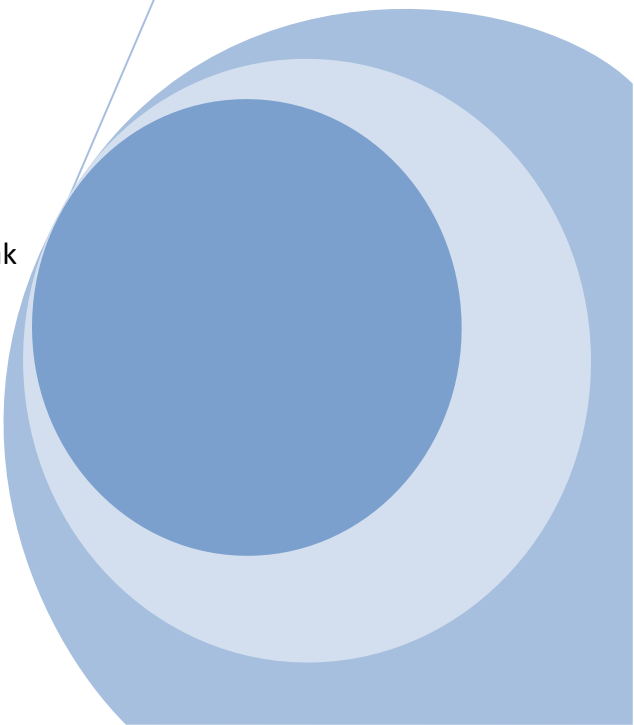
OpenJ

Linköpings universitet

Språkdokumentation för kursen TDP019 Projekt: Datorspråk

Joacim Wiell – joavi869@student.liu.se
Oscar Zanton – oscza478@student.liu.se

2010-05-22



Innehållsförteckning

1. Inledning

Vi går programmet Innovativ programmering och har som projektarbete under andra terminen gjort programspråket OpenJ. Målet (direkt hämtat från LiTH:s studiehandbok[1] för kursen TDP019 Projekt: Datorspråk) med detta projekt har varit att vi efter kursen ska kunna:

- konstruera ett mindre datorspråk,
- diskutera och motivera designval i det egna datorspråket med utgångspunkt i teori och egna erfarenheter
- implementera verktyg (interpretator, kompilator, etc.) för det egna datorspråket
- formulera teknisk dokumentation av det egna datorspråket

Språket kombinerar egenskaper från Python såsom dynamisk typning och bl.a. strukturen för funktioner från C++. Alla variabler har en räckvidd och det finns globala variabler som kan nås från alla nivåer i programmet. Språket riktar sig till nybörjare och personer som har tidigare erfarenhet av programmering.

[1] http://kdb-5.liu.se/liu/lith/studiehandboken/svkursplan.lasso?&k_budget_year=2010&k_kurskod=TDP019

2. Användarhandledning

I detta kapitel kommer det att beskrivas hur det går till att installera och använda OpenJ, dessutom beskrivs hur det går till att programmera i OpenJ.

2.1 Installation

Ladda ner filen ”OpenJ_v0.1.zip” från http://dl.dropbox.com/u/3487827/OpenJ_v0.1.zip

2.1.1 Allmänna krav

Ruby v1.8.7 eller senare (endast testat med v1.8.7).
En text editor behövs för att programmera i. Vi rekommenderar Emacs, gedit eller Scite på Linux och Scite (medföljer vid installationen av Ruby) eller Notepad++ på Windows.

2.1.2 Linux & Mac OS X

Extrahera filerna ur arkivet och placera dessa på ett lämpligt ställe t.ex. ”`~/OpenJ`”.

För att göra OpenJ mer användbar under Linux följ följande punkter:

Öppna filen ”OpenJ.rb” med ett textprogram (t.ex. emacs, Scite eller gedit).
Redigera den 6:e raden i filen (`install_directory = ""`). Lägg till sökvägen till OpenJ innanför citattecknen (ex. `"~/OpenJ/"`). OBS! Glöm inte bort att använda ett sista ”/” i sökvägen!
Kör sedan kommandot ”`chmod a+x`” på filen ”OpenJ.rb” (”`chmod a+x OpenJ.rb`”) vilket gör filen körbar.

Skapa sedan en filen `.bash_aliases` i roten av din hemkatalog (”`~/`”).
Lägg till följande sträng i `.bash_aliases`:
”`alias OpenJ='<install_path>/OpenJ.rb'`” (utan det första och sista citattecknet, <install_path> ska bytas ut mot sökvägen t.ex. ”`~/OpenJ/`”).
Spara filen och stäng. Nästa gång du loggar in igen kommer kommandot OpenJ finnas tillgängligt.

Nu kan du köra program i Linux genom att öppna ett terminalfönster och skriva

”`OpenJ <fil att köra>`” (ex. `OpenJ test.oj`).

2.1.3 Windows

Extrahera filerna ur arkivet och placera dessa på ett lämpligt ställ t.ex. ”`C:\OpenJ`”.

För att göra OpenJ mer användbar under Windows följ följande punkter:

Öppna filen ”OpenJ.rb” med ett textprogram (t.ex. Scite eller notepad).
Redigera den 6:e raden i filen (`install_directory = ""`). Lägg till sökvägen till OpenJ innanför citattecknen (ex. ”`C:\\OpenJ\\`”). OBS! Det ska vara ”`\\`” mellan mappnamnen, glöm inte heller bort att det ska vara ”`\\`” i slutet av sökvägen!

Lägg sedan till sökvägen i miljövariabler ”`Path`”:
Startmenyn > Kontrollpanelen > System, klicka på Avancerade Systeminställningar.
Gå till fliken Avancerat, klicka på Miljövariabler. Under Systemvariabler finns ”`Path`”.
Dubbelklicka på denna och i slutet av strängen lägg till ”;<sökväg>” (ex. ”`;C:\OpenJ\`”).

Nu kan du köra program i OpenJ genom att öppna ett terminalfönster och skriva

”OpenJ <fil att köra>” (ex. ”`OpenJ test.oj`”).

2.1 Att komma igång

Likt C++ läser OpenJ in vilka instruktioner som skall utföras från en fil men skillnaden är att OpenJ inte skapar en körbarfil utan det är OpenJ som exekverar/kör instruktionerna. Det kallas att man kör en skriptfil i en virtuell maskin.

2.2 Ett första exempel

Detta första lilla exempel är ett väldigt simpelt program som endast kommer att skriva ut ”`Hello World`” på skärmen.
Exempel:

```
def main()
{
    print ”Hello World!”
}
```

Nu kommer en beskrivning rad för rad om vad som händer i koden:
`def main()`
Skapar en funktion med namnet `main`, det är viktigt att det finns en funktion i programmet som heter main, annars kommer inte programmet att köras.
`{`
Den första klammerparantesen talar om för OpenJ att koden som kommer efter ska tillhöra den konstruktionen som specificerats tidigare (i detta fall funktionen main).
`print ”Hello World!”`
Ordet `print` är ett nyckelord och kommer att skriva ut något på skärmen, allt innanför citattecknen kommer att tolkas som en sträng av OpenJ. Kommandot `print` kommer alltså att skriva ut strängen `Hello World!` på skärmen.

2.3 Variabler

Man kan säga att variabler är en behållare där man lagrar data som man använder för att hantera data i programmet. När man ger variabeln ett namn kallas det att den deklareras. I OpenJ måste variabler börja med antingen en bokstav eller ett understrykningstecken för att vara ett giltigt variabelnamn.

Exempel på giltiga variabelnamn:

```
var
var1
_var
```

För att tilldela en variabel, lagra data i den, så skriver man variabelnamnet först följt av = tecken och data man vill ge den.

Exempel på en tilldelning där heltalet 1 lagras i variabeln iTal:

```
iTal = 1
```

2.4 Räckvidd

Med en variabels räckvidd menar man var i programmet som variabeln är åtkomlig. Variabler gäller i den funktion eller `for`-loop som den definierad. Det kommer mer om variabelräckvidd i avsnitt 2.9.2 For-loopen och 2.10 Funktioner.

2.5 Globalvariabel

När man vill kunna nå data från vilken del av programmet som helst använder man sig av något som kallas globalvariabel. Man bör dock vara restriktiv med en sådan variabel och endast använda den då man endast vill lagra data som ska kunna ändras oavsett var man befinner sig i programmet. En global variabel deklareras med ett `@` tecken framför namnet.

Exempel på en globalvariabel som innehåller texten `Hello world!`:

```
@global_var = ”Hello world!”
```

2.6 Typer

I detta avsnitt kommer alla typer som finns tillgängliga beskrivas.

2.6.1 Heltal

Heltal är tal som är positiva eller negativa utan decimaler. För att tilldela en variabel ett heltal skrivs

Exempel med variabelnamnet `iTal`:

```
iTal = 1
```

2.6.2 Flyttal

Flyttal är tal som positiva eller negativa tal med decimaler.

Exempel med variabelnamnet `fTal`:

```
fTal = 1.0
```

2.6.3 Sträng

En sträng är ett fält som lagrar tecken/meddelanden som framför allt är tänkt att kunna läsas av människor. En sträng är flera tecken som skrivs i en följd och innesluts av antingen citattecken eller enkla citattecken. Det går inte att kombinera dessa utan man måste välja antingen eller.

Exempel med variabeln `sStr` först med citattecken sedan med den alternativa metoden enkla citattecken:

```
sStr = ”Hello World!”
sStr = ’Hello World!’
```

2.6.4 Array/fält

Array är en typ som kan innehålla flera variabler samtidigt av flera typer. Informationen som finns lagrad i ett fält nås genom index som börjar med `0`, som är den första positionen i fältet. För att tilldela data till ett fält så innesluts den med halkparenteser och varje post separeras med kommatecken.

Exempel med variabelnamnet `aArr` som innehåller två heltal och en sträng:

```
aArr = [1, 2, ”Hello”]
```

Exemplet ovan är första positionen i fältet heltalet `1` och har positionen `0`, som är den första, och strängen `Hello` har positionen `2`.

2.6.5 Hashtabell

Hashtabell är en typ som skapar associativa fält med hjälp av en nyckel och ett värde. Liksom en array kan en hashtabell lagra flera variabler av flera typer samtidigt men den stora skillnaden är istället för att använda index för att bearbeta data så används en nyckel. För att OpenJ ska känna igen att det är en hashtabell man vill använda omsluts den data som tilldelas till hash-tabellen av klammerparentes. I klammerparentesen så separeras varje post av ett kommatecken och varje post består av en nyckel, kolon och ett värde.

Exempel här skapar vi en hashtabell med namnet hHash och tilldelar den två associativa fält med nycklarna `first` och `22`, `first` har värdet `1` och `22` har värdet `Hello`:

```
hHash = {’first’:1, 22:’Hello’}
```

Eftersom att `first` och `Hello` är strängar omsluts dessa här av enkla citattecken, vanliga citattecken fungerar lika bra.

2.6.6 Boolesk

En boolesk datatyp innehåller antingen är sant eller falsk-värde och OpenJ använder de engelska orden `true` för sann och `false` för falsk. Datatypen kan användas när man vill ha en operation med villkor i programmet.

Exempel med variabelnamnet `bVar` som tilldelas ett sant värde.

```
bVar = true
```

2.7 Operatorer

Operatorer talar om hur variablerna ska samverka och för att ge ett resultat. Vid aritmetiska operationer med multiplikation och division evalueras dessa före addition och subtraktion om inte additionen eller subtraktionen står inom parentes, då evalueras detta först.

Tabell 1, Aritmetiska operatorer

Namn	Symbol	Exempel
Ökning med 1	++	`i++` betyder samma sak som `i = (i + 1)`
Minskning med 1	--	`i--` betyder samma sak som `i = (i - 1)`
Multiplikation	*	`2 * 2` ger 4
Division	/	`10 / 2` ger 5
Addition	+	`1 + 1` ger 2
Subtraktion	-	`3 - 2` ger 1

Tabell 2, Logiska operatorer

Namn	*Symbol*	I exemplen nedan `bVar1 = true` och `bVar2 = false`
Inte, negerar uttrycket.	Not, not	`not bVar1` ger ”false”
Och, båda uttrycken måste vara sanna	And, and	`bVar1 and bVar1` ger ”true”
		`bVar1 and bVar2` ger ”false”
Exklusivt eller, om ett av uttrycken är sant är hela satsen sann annars inte.	Xor, xor	`bVar1 xor bVar1` ger ”false”
		`bVar1 xor bVar2` ger ”true”
Eller, ett av uttrycken måste vara sanna.	Or, or	`bVar1 or bVar2` ger ”true”

Tabell 3, Jämförelse operatorer

Namn	*Symbol*	I exemplen nedan `iTal1 = 1` och `iTal2 = 2`
Mindre än	<	`iTal1 < iTal2` ger ”true”
Större än	>	`iTal1 > iTal2` ger ”false”
Lika med	==	`iTal1 == iTal1` ger ”false”
Större än eller lika med	>=	`iTal1 >= iTal1` ger ”false”
Mindre än eller lika med	<=	`iTal1 <= iTal2` ger ”true”
Skilt från, ej lika med	!=	`iTal1 != iTal2` ger ”true”

	<>	`iTal1` **<>** `iTal2` ger ”true”

2.8 Kontrollsats

Kontrollsatser används för att testa om ett värde i programmet är sant eller inte. En metod för att testa ett villkor är en `if`-sats som testar två värden eller uttryck med en jämförelse operator mellan. Beroende på utfallet av den jämförelsen kan man exempelvis köra en kod sats. `If`-satsen börjar med `if` följt av parenteser där de villkor man vill evaluera skrivs.

Exempel som använder variablerna `iTal1` som tilldelats `1` och `iTal2` som tilldelats `2`:

```
if(iTal1 < iTal2)
{
  print iTal1
}
```

Eftersom att `iTal1` är mindre än `iTal2` i exemplet ovan kommer uppfylls villkoret och det värde som variabeln `iTal1` har kommer skrivas ut. Det går även att göra så att en program-kod körs om villkoret inte uppfylls och det görs med kommandot `else`. Det går även att lägga till ett andra villkor om efter `if` och det heter `elseif`.

Exempel som bygger ut det förra exemplet där vi först testar om `iTal1` är större än `iTal2` och detta är sant skriver ut `iTal1`. Annars testar vi om `iTal2` är större än `iTal3` och skriver då ut `iTal2`. Om båda dessa villkor inte är sanna skrivs `iTal3` ut.

```
if(iTal1 > iTal2)
{
  print iTal1
}
elseif(iTal2 > iTal3)
{
  print iTal2
}
else
{
  print iTal3
}
```

Först testas alltså det villkor som står i parenteserna efter `if`, om detta inte är sant testas det villkor som står i parentesen efter `elseif` och om båda dessa villkor skulle vara falska körs koden som finns i `else`.

2.9 Loopar

En loop är en programkod som repeteras så länge som ett villkor uppfylls. I OpenJ finns det två olika loopar att använda sig av och dessa är `while` och `for`. Om man vill avbryta en loop kan man exempelvis med en `if`-sats lägga att när detta är uppfyllt körs kommandot `break` och loopen slutar omedelbart.

2.9.1 While-loopen

Om man vill repetera något där bara ett villkor ska vara uppfyllt är `while`-loopen ett bra alternativ. `While`-loopen skrivs enligt följande och satsen kommer köras så länge som villkoret är uppfyllt, skriver man exempelvis `while(true)` kommer satsen repeteras i oändlighet:

```
While(villkor)
{
  sats
}
```

Exempel om vi vill skriva ut talen `1` till `10` på skärmen.

```
a = 1
while(a < 11)
{
  print a
  a++
}
```

Här använder vi oss av variabeln a som initialt sätts till `1` utanför `while`-loopen. Sedan testar vi om `a` är mindre än `11` och så länge som det villkoret är uppfyllt kommer satsen med `print a` som skriver ut värdet för variabel `a` och sedan adderar på heltalet `1` till `a`.

2.9.2 For-loopen

Om man vet hur många gånger en sats ska repeteras kan `for`-loopen vara ett bättre alternativ än `while`-loopen. Syntaxen för `for`-loopen är:

```
For(initiering; villkor; ändring)
{
  sats
}
```

Först körs initieringen följt av villkoret och om villkoret är uppfyllt så körs satsen och sedan ändringen. Det är valfritt om man vill använda initiering, villkor eller ändrig i `for`-loopen, den fungerar utan alla dessa eller med en blandning av de olika. Om vi kör samma exempel som för `while`-loopen där vi vill skriva ut talen `1` till `10` på skärmen.

```
For(i=1;i<11;i++)
{
  print i
}
```

Till skillnad från `while`-loopen så initierar vi variabeln `i` och sätter den till `1` först när `for`-loopen körs. Sedan testar vi villkoret om `i` är mindre än `11` och om det är sant så körs `print i` och värdet för variabeln `i` skrivs ut. Efter det att villkoret testats och satsen körts så gör vi en ändrig där vi ökar variabeln `i` med `1`.

Alla variabler som deklareras inuti en `for`-loop kommer ha en räckvidd som bara gäller i den loopen. Det betyder att när loopen avslutas blir variabeln otillgänglig.

2.10 Funktioner

OpenJ använder sig av funktioner som enkelt sagt är en del av ett program som kan anropas från andra delar av programmet för att utföra en viss uppgift helt oberoende av vad andra delar gör. Syntaxen för att skriva en funktion är:

```
def funktion(parameter1, parameter2, parametern)
{
  sats
}
```

Parametrarna är valfria och behövs inte för att deklarera en funktion utan behövs endast om funktionen ska kunna skicka in ett värde i funktionen exempel:

```
def funk1(iTal)
{
  print iTal
}
```

Vad denna funktion gör är att ta in ett värde i parametern `iTal` och sedan skriver ut den. För att ”ropa på” eller köra en funktion så skriver man funktionsnamnet följt av parenteser där man anger eventuella variabler som ska skickas med.

Exempel som anropar funktionen `funk1` och där funktionen skriver ut parametern:

```
funk1(42)
```

Funktionerna kan även ge ett eller flera returvärden dvs. returnera resultat till variabler. *Exempel* på detta:

```
def funk2(iTal1, iTal2)
{
  return iTal1 + iTal2
}
```

Funktionen `funk2` tar in två parametrar och ger tillbaka resultatet av dessa.

Exempel på hur ett anrop kan ske till variabeln `iResultat`:

```
iResultat = funk2(2, 6)
```

`iResultat` kommer i det här fallet vara ett heltal som har värdet `8`.

Det går även att returnera flera resultat från en funktion till flera variabler.

Exempel på detta:

```
def funk3(iTal1, iTal2)
{
  return iTal1, iTal2
}
```

och vid funktionsanropet skriver man då med `iTal3` och `iTal4` som variabler:

```
iTal3, iTal4 = funk3(5, 6)
```

Resultatet av detta är att `iTal3` kommer få värdet `5` och `iTal4` kommer få värdet `6`.

För att skriva ett program i OpenJ måste funktionen `main` finnas med, det är den funktionen som är huvudfunktionen där programmet börjar köras ifrån. `main`-funktionen har inget returvärde och inga parametrar och skrivs enligt:

```
def main()
{
  sats
}
```

För att komplettera koden ovan till ett körbart program skulle det då bli:

```
def main()
{
  funk1(42)
}

def funk1(iTal)
{
  print iTal
}
```

Det går att nästla funktioner i OpenJ, vilket betyder att det går att definiera funktioner inuti andra funktioner. Trots detta blir funktionen global alltså att den kan kallas varifrån som helst i programmet. Nästling har inte heller någon betydelse för variabelräckvidden i den nästlade funktionen eftersom OpenJ använder dynamisk bindning på variablerna. Dynamisk bindning betyder att värdena sätts på variablerna under körning och inte under kompilering. En nästlad

funktion kommer ha tillgång till variabler från funktionen där den kallas ifrån men det gäller inte tvärt om. Alltså att funktionen utanför som kallar på den andra funktionen har inte tillgång till variabler inuti den nästlade funktionen.

2.11 Kommentarer

När man skriver kod är det mycket bra att kunna skriva kommentarer över exempelvis vad en funktion gör och för att det ska bli lättare att hitta i koden. Detta gör det lättare att underhålla koden och för att andra programmerare lättare ska kunna sätta sig in vad någon annan gjort innan. Vid programkörningen i OpenJ ignoreras allt som står som kommentarer.

I OpenJ finns det två typer av kommentarer, enradskommentarer som börjar med två snedstreck och det andra sättet är flerradskommentarer där det blocket med kommentarer börjar med snedstreck och stjärna och slutar med stjärna och snedstreck.

Exempel på enradskommentar:

```
// Allt på efter här kommer att ignoreras.
```

Exempel på flerradskommentarer:

```
/* Allt jag skriver här
och här
och här ignoreras vid körning
*/
```

2.12 Input

OpenJ kan ta emot data från användaren och exempelvis tilldela en variabel vid körning med något som användaren skriver. Programmet ger då en prompt och väntar tills användaren trycker vagnretur innan det fortsätter.

Exempel som tilldelar variabeln `sVar`:

```
sVar = input
```

2.13 Reserverade ord

I OpenJ finns det vissa ord som bara får användas av OpenJ. Med detta menas att inga variabler eller funktioner får ha något av orden i tabellen nedan.

Tabell 4, Reserverade ord

If/if	Elseif/elseif/ElseIf/elseIf	Else/else	While/while
Not/not	And/and	Or/or	break
def	input	return	true/false

3. Systemdokumentation

I detta kapitel kommer uppbyggnaden av OpenJ att beskrivas, vilka delar det innehåller och vad de olika delarna gör.

3.1 Start

För att köra program i OpenJ måste man skicka med ett filnamn till en fil som innehåller kod som ett argument till filen ”OpenJ.rb”, OpenJ kommer då att kontrollera att filen existerar och sedan läsa in den.

3.2 Tokens och parsing

Efter filinläsningen skett kommer det skapas en instans av klassen OpenJ. Inuti klassen OpenJ finns alla regler som används för att hitta tokens och de grammatiska regler som tokens ska matchas mot. Efter detta skapas en instans av klassen Parse där reglerna för tokens och grammatik används. När en bit kod matchas mot en regel för token som t.ex. ”`token(/\d+\.\d+/)`[2] `{|m| m.to_f }`[3]”. Denna matchning letar efter en token med en eller flera siffror följt av en ”.” och om sådan token har hittats körs kodblocket och med den inlästa strängen konverteras till ett flyttal (`Float`). När alla ord och tecken har blivit tokens läser instansen in alla grammatiska regler (se Grammatik), skapas den en instans av klassen Rule. Instansen av Rule försöker att matcha en regel för en eller flera tokens, om inte det stämmer med regeln försöker den matcha mot nästa regel annars körs koden i kodblocket efter.
Exempel på en regel för matchning är följande kod:

```
rule :numbers do
     match(Integer) {|int| IntegerNode.new(int)}
     match(Float) {|float| FloatNode.new(float)}
end
```

Där tokens identifierade med ”`token(/\d+\.\d+/) {|m| m.to_f}`” kommer att hamna och skapa en nod av typen `FloatNode` som kommer att innehålla det värdet tokens kodblock returnerar.
Kodblocken för reglerna består oftast av att skapa en instans av en av noderna, som sedan kommer att länkas ihop vilket i slutet ger ett syntaxträd.

[2] Regel för en token
[3] Kodblock

3.3 Noder

Alla noder som finns har oftast två funktioner. Dessa är `initialize` och `eval`. `initialize` är den funktion som körs när noden skapas, dess uppgift är oftast ta argument och sätta instansvariabler[4] i noden.

Funktionen `eval` har som uppgift att köra en kod som är specifik för noden. Denna kod kommer samtidigt att kalla på `eval`-funktionen i eventuella noder som finns under sig själv. *Exempel* på en nod:

```
class StmtsNode

  def initialize(stmt, stmts=nil)
    @stmt = stmt
    @stmts = stmts
  end

  def eval
    return_value = @stmt.eval()
    if (return_value.class == ReturnNode or
        return_value.class == BreakNode)
      return return_value
    end
    if(@stmts != nil)
      return_value = @stmts.eval()
    end
    return return_value
  end
end
```

I exemplet visas noden `StmtsNode` (förkortning StatmentsNode) som används för att kunna ha flera statments efter varandra. Funktionen `initialize` tar emot 2 argument, `stmt` och `stmts`, där `stmt` kommer var ett statement och `stmts` kommer att vara `nil` (inget värde) eller en ny `StmtsNode`. De två tilldelningssatserna kommer att sätta dessa värden till två instansvariabler.

Det första som händer i funktionen `eval` är att det körs `eval` på den nod som finns i `@stmt` och lägga returvärdet i variablen `return_value`. Sedan kontrolleras värdet på `return_value`, om det ligger en nod av typerna `ReturnNode` eller `BreakNode` kommer värdet att returneras till noden som kallade på denna `StmtsNode`, om det inte ligger en `ReturnNode` eller `BreakNode` kommer det kontrolleras att `@stmts` inte är `nil`. Om det inte är `nil` kommer `eval` kallas på den nod som ligger i `@stmts` och det returvärdet från den noden kommer att läggas i `return_value` som sedan kommer att returneras.

[4] Variabler som är tillgängliga för i hela den instans som skapats av klassen.

3.4 Syntaxträd

I vår implementation byggs syntaxträdet upp av noder, där varje nod kan innehålla fler noder nedanför sig själv i trädet.
Exempel på detta är följande programkod:

```
def main()
{
      x = 0
      print x
}
```

Denna kod kommer att genera syntaxträd på bild 1.

För att bygga upp syntaxträdet används de grammatiska reglerna (se Grammatik) för att matcha de olika elementen i koden och för att skapa rätt nod för varje regel.

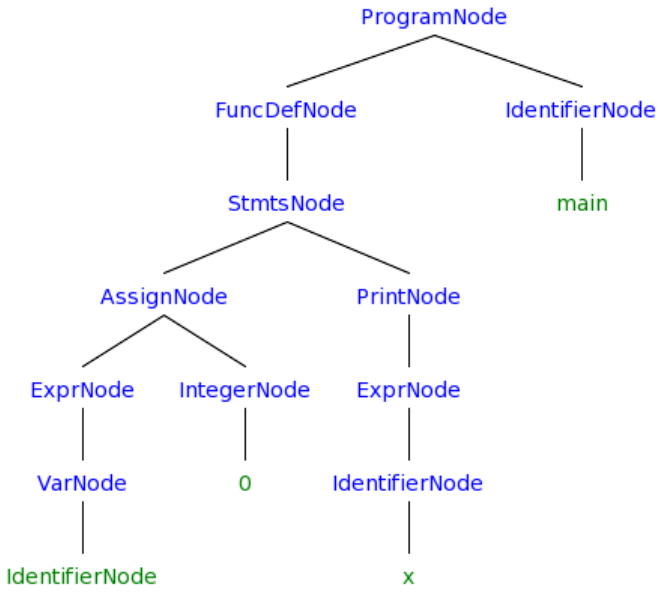


Bild 1, Syntaxträd

3.5 Evaluering

Efter att syntaxträdet är uppbyggt kallar översta noden i trädet som alltid är en ProgramNode på funktionen ”main” som måste finnas med i alla program.
Alla noderna i syntaxträdet kommer att kalla på funktionen eval (som finns med i alla noderna) i noden under. Sista noden i kedjan kommer att skicka tillbaka sitt resultat högre upp tills alla noderna har körts.

3.6 Använda algoritmer

Eftersom vårt språk körs i Ruby kan vi använda oss av Rubys algoritmer. OpenJ bygger direkt på användningen av t.ex. strängar, tal, arrayer. OpenJ använder sig också av Rubys funktioner för att påverka dessa typer, som att addera två heltal där Rubys additionsmetod för `Fixnum` eller `Bignum` används. Det som inte är lätt att se är att OpenJ förlitar sig på Rubys minneshantering. Att objekt som inte behövs längre tas bort utan att användaren behöver göra något.

3.7 Kodstandard

OpenJ tvingar inte användaren att använda indentering i koden, men som alltid uppmanas kodare att använda detta för att koden blir lättare att läsa. Alla identifierare i OpenJ måste börja med ”_”, a-z eller A-Z för att sedan ha noll eller fler av följande tecken ”_”, a-z, A-Z eller 0-9. Detta betyder att alla funktions- och variabelnamn måste ha detta utseende men detta gäller dock inte för globala variabler som måste börjar med ett ”@” men i övrigt ser ut som

funktions- och variabelnamn. Se tabell 5 för exempel på vilka variabelnamn som är godkända.

Tabell 5, Variabelnamn

	Variabel-/Funktionsnamn			Globala variabelnamn		
Godkända	a1	_a	_1	@a	@_a	@_1
Ej godkända	1_a	2__	4b	@1_a	@2__@	@4b

3.8 Klasser

I detta avsnitt kommer alla klasser som finns i OpenJ att listas samt ge information om vilka klasser som ärver från varandra, vilka klasser som använder varandra och vad klassen används till.

3.8.1 OpenJ

Använder: Parser och Variables.
Klassen OpenJ innehåller alla regler för token och grammatik.

3.8.2 Parser

Använder: ParseError och Rule.
Klassen Parser innehåller funktioner för att skapa tokens och parsa tokens mot grammatiska regler.

3.8.3 ParseError

Ärver från: RuntimeError.
ParseError används för att kasta felmeddelande som uppstår vid problem med parsingen i Parser.

3.8.4 Variables

Klassen Variables används för att lägga till och hämta variabler ur variabellistan och hålla reda på vilka variabler som tillhör vilken räckvidd.

3.8.5 AritmExprNode

Noden AritmExprNode används vid för att utföra aritmetiska beräkningar som t.ex. `1 + 1`.

3.8.6 ArrayNode

ArrayNode skapar ett objekt av datatypen Array.

3.8.7 AssignNode
Använder: OpenJ
AssignNode tilldelar variabler värde t.ex. `x = 1`.

3.8.8 AssignReturnNode
Använder: OpenJ
AssignReturnNode tilldelar returvärde från funktionsanrop t.ex. `x = main()`

3.8.9 BoolNode
BoolNode skapar ett objekt av datatypen Boolean (Boolesktvärde).

3.8.10 BreakNode
BreakNode används för avbryta en `While`- eller `For`–loop, avslutar även `if`-satser och funktioner om den inte finns inuti en `While`-loop.

3.8.11 CompExprNode
Noden CompExprNode används för att jämföra variabler med jämförelseoperatorer
t.ex. `1 < 2`.

3.8.12 FloatNode
FloatNode skapar ett objekt av datatypen Float (flyttal).

3.8.13 ForNode
ForNode används för att skapar en `For`-loop som körs till att kontrollvärdet är uppfyllt eller tills den bryts av `break`.

3.8.14 FuncCallArgNode
Noden FuncCallArgNode innehåller ett argument till ett funktionsanrop.

3.8.15 FuncCallArgListNode
Noden FuncCallArgListNode finns för att det ska kunna gå att skicka mer än ett argument till ett funktionsanrop.

3.8.16 FuncCallNode
Använder: Variables
Klassen FuncCallNode används för att kallar på en funktion t.ex. `test()`

3.8.17 FuncDefsNode
Noden FuncDefsNode finns för att det ska gå att ha flera funktionsdefinitioner i samma program.

3.8.18 FuncDefNode
Använder: Variables
FuncDefNode innehåller en funktionsdefinition

3.8.19 FuncDefParamNode
FuncDefParamNode innehåller en parameter till en funktion.

3.8.20 FuncDefParamListNode
Noden FuncDefParamListNode finns för att det ska kunna gå att skicka mer än en inparameter till en funktion.

3.8.21 GlobalVarNode
GlobalVarNode innehåller en globalvariabel.

3.8.22 HashNode
HashNode skapar ett objekt av datatypen Hash (Hashtabell).

3.8.23 IdentifierNode
IdentifierNode innehåller ett variabel-, globalvariabel- eller ett funktionsnamn.

3.8.24 IfNode
IfNode används för att skapa en `if`-sats, används även för `elseif` och `else` grenar om dessa finns.

3.8.25 InputNode
InputNode tar emot indata från användaren via tangentbordet.

3.8.26 IntegerNode
IntegerNode skapar ett objekt av datatypen Integer(heltal).

3.8.27 LogicExprNode
LogicalExprNode används för att beräkna ett logisktuttryck t.ex. ”`x and y`”.

3.8.28 NegateNode
NegateNode används för att negera tal och variabler t.ex. `-10` eller `-a`.

3.8.29 NullNode
Används när något är `nil` (inte har något värde alls).

3.8.30 PrintNode
PrintNode skriver ut ett utryck som står efter. Om det inte finns något uttryck efter skrivs en tom rad ut på skärmen.

3.8.31 ProgramNode
Använder: Variables
ProgramNode är starten på programmet, högsta noden i trädet.

3.8.32 ReturnExprNode
ReturnExprNode innehåller ett värde som ska returneras av ReturnNode

3.8.33 ReturnExprsNode
Noden ReturnExprsNode finns för att det ska kunna gå att skicka tillbaka mer än ett värde via ReturnNode.

3.8.34 ReturnNode
ReturnNode avbryter en funktion och returnerar eventuellt uttryck efter.

3.8.35 StmtsNode
StmtsNode finns för att det ska kunna gå att ha flera stmts efter varandra.

3.8.36 StringNode
StringNode skapar ett objekt av datatypen String (sträng).

3.8.37 TermNode
Noden TermNode gör att det kan finnas flera termer efter varandra.

3.8.38 VarNode
VarNode innehåller en variabel.

3.8.39 VarsNode

Noden VarsNode finns för att det ska vara möjligt att för AssignReturnNode ska kunna tilldela flera värde.

3.8.40 WhileNode

Använder: OpenJ
WhileNode skapar en while-loop som körs tills villkoret är uppfyllt eller tills den avbryts av break.

4. Erfarenhet och reflektion

I början av projektet gick det trögt. Vi fick inte riktigt något grepp om hur allt skulle implementeras i rdparsern eller i Ruby. Men mot slutet när man verkligen hade kommit in i projektet började allt rulla på. Fel på regler och syntaxträdets uppbyggnad kunde nästan undvikas helt, men om det blev ett fel visste vi ungefär var och varför. I början fick man sitta länge för att identifiera varför samma fel uppstod. Det tog ett bra tag innan vi kom igång med implementationen, i början av kursen kändes det att vi hade hur gott om tid som helst men mot slutet kände vi hade för lite tid istället. Det fanns de sakerna som vi verkligen önskade att vi hade tid att implementera, men i övrigt tycker vi att arbetet har gått bra och framåt hela tiden.
Vi har implementerat det mesta av vad vi hade med i vår första språkdefinition. Det som inte fullföljdes var att implementera valfri statiskbindning av variabeltyp för variabler. Men det ser vi inte som ett måste att ha i språket. Vi har inte heller implementerat arrayer och hashtabeller mer än att de finns tillgängliga att skapa, det finns dock igen funktion för att hämta ut ett specifikt värde ur dessa eller påverka dem på annat sätt än att ersätta hela variabeln med ett nytt värde.
Vi upptäckte när vi försökte att evaluera koden att vi hade missat att lägga till en viktig nod i början, toppnoden i trädet, därför la vi till noden ProgramNode som startar evalueringen nedåt i noden. Utöver detta var vi också tvungna att lägga till flera noder för att kunna ha många av samma nod efter varandra, t.ex. StmtsNode som finns för att det ska kunna komma många Statements efter varandra istället för bara en.
Vi hade ett problem med return i språket, om man skrev return utanför en funktion i programmet avslutades hela programmet där. Det löste vi genom att införa att programmet var tvungen att bestå av en eller fler funktioner som C++, alltså att det inte bara går att skriva flera satser av kod och säg att det är ett program utan att man måste definiera en funktion som heter main i programmet och som körs först när programmet evalueras.
Om vi kommer att bygga fler datorspråk kommer vi att ha en heldel av erfarenheter med oss till det projektet. Viktigaste är att det inte är lika svårt man tror att det är i början, utan när man väl har kommit igång med grammatiken och implementationen kommer det mesta att ge sig själv när man testkör programspråket.

5. Programkod

Här visas all programkod från OpenJ, den är uppdelad efter vilken av tre filer den ligger i. Dessa filer är 5.1 OpenJ.rb, 5.2 rdparse.rb och 5.3 Nodes.rb.

5.1 OpenJ.rb

```
#!/usr/bin/env ruby

#       ONLY CHANGE HERE!
##########################################
#
install_directory = ""
#
##########################################

require 'fileutils'
if (install_directory == "")
  require 'rdparse'
  require 'Nodes'
else
  require install_directory + 'rdparse'
  require install_directory + 'Nodes'
end


unless ARGV.length == 1
  puts "This program requires one parameter, the file you want to run!"
  puts "Usage OpenJ file.oj"
  exit
end

class OpenJ

  def initialize
    OpenJ.init()
    ################ Create Regexpes here! ################
    create_keyword_list()
    arr_regexp = "^\\[((\\d*|((\\\"|\\\')\\w*(\\\"|\\\')))\\s*\\,*\\s*)*\\]$"
    array_regexp = create_regexp(arr_regexp)

    hash_regexp = "^\\{((((\\\"|\\\')\\w*(\\\"|\\\'))|\\d*)\\s*\\:\\s*"
    hash_regexp += "(((\\\"|\\\')\\w*(\\\"|\\\'))|\\d*)\\s*\\,*\\s*)*\\}$"
    hash_regexp = create_regexp(hash_regexp)

    whitespace_regexp = create_regexp("\\s")
    float_regexp = create_regexp("\\d+\\.\\d+")
    int_regexp = create_regexp("\\d+")
    online_comment_regexp = create_regexp("\\/\\/.*")
    multiline_comment_regexp = create_regexp("\\/\\*[\\w\\W\\s]*\\*\\/")
    bool_regexp = create_regexp("(true|false)")
```

```
  word_char_regexp = create_regexp("\\w+")
  string_regexp = create_regexp("(\\\".+\\\"|\\\"\\\"|\\'.+\\'|\\'\\')")
  inc_dec_regexp = create_regexp("(\\+\\+|\\-\\-)")
  operator_regexp = create_regexp("(\\+|\\-
|\\*|\\/|\\=\\=|\\<\\=|\\>\\=|\\<\\>|\\!\\=)")
  variable_regexp = create_regexp("@?[_a-zA-Z]+[_a-zA-Z0-9]*")
  all_chars = create_regexp(".")

  globalvar_regexp = create_regexp("@[_a-zA-Z]+[_a-zA-Z0-9]*",@keywords)
  var_regexp = create_regexp("[_a-zA-Z]+[_a-zA-Z0-9]*",@keywords)
  and_regexp = create_regexp("[Aa]nd")
  not_regexp = create_regexp("[Nn]ot")
  xor_regexp = create_regexp("[Xx]or")
  or_regexp = create_regexp("[Oo]r")
  for_regexp = create_regexp("[Ff]or")
  if_regexp = create_regexp("[Ii]f")
  elseif_regexp = create_regexp("[Ee]lse[Ii]f")
  else_regexp = create_regexp("[Ee]lse")
  while_regexp = create_regexp("[Ww]hile")
  ##############################################

  @diceParser = Parser.new("OpenJ") do
    token(whitespace_regexp)
    token(float_regexp) {|m| m.to_f }
    token(int_regexp) {|m| m.to_i }
    token(online_comment_regexp) # token for oneline comment
    token(multiline_comment_regexp) # token for multiline comment
    token(bool_regexp) {|m| m}
    token(word_char_regexp) {|m| m} # token for any word character
    token(string_regexp) {|m| m} # token for string
    token(inc_dec_regexp) {|m| m} #token for incrament and decrament (++ --)
    token(operator_regexp) {|m| m}
    token(variable_regexp) {|m| m} # token for global variable and variables
    token(array_regexp) {|m| eval(m)} # token for array
    token(hash_regexp) {|m| eval(m.gsub(":","=>"))}# token for hash
    token(all_chars) {|m| m }

    start :program do
      match(:funcDefs) {|a| ProgramNode.new(a)
        nil}
    end

    rule :funcDefs do
      match(:funcDef, :funcDefs) {|func_def, func_defs|
        FuncDefsNode.new(func_def,func_defs)}
      match(:funcDef)
    end

    rule :stmts do
      match(:stmt, :stmts) {|stmt, stmts| StmtsNode.new(stmt,stmts)}
      match(:stmt) {|stmt| StmtsNode.new(stmt)}
    end

    rule :stmt do
      match(:break)
```

```
  match(:special)
  match(:return)
  match(:forStmt)
  match(:ifStmt)
  match(:whileStmt)
  match(:funcDef)
  match(:funcCall)
  match(:assign)
end

rule :break do
  match("break") {BreakNode.new()}
end

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

rule :returnExprs do
  match(:returnExpr, ',', :returnExprs) {|expr, _, exprs|
    ReturnExprsNode.new(expr,exprs)}
  match(:returnExpr) {|expr| ReturnExprsNode.new(expr)}
end

rule :returnExpr do
  match(:expr) {|expr| ReturnExprNode.new(expr)}
end

rule :special do
  match(:print)
end

rule :input do
  match('input') {|input| InputNode.new(input)}
end

rule :print do
  match('print', :expr) {|_, expr| PrintNode.new(expr)}
  match('print') {PrintNode.new()}
end

rule :expr do
  match(:addiExpr)
  match(:term)
end

rule :compExpr do
  match(:addiExpr)
  match(:boolean)
  match(:compExpr, :compOperator, :compExpr) {|expr1, op, expr2|
    CompExprNode.new(expr1,op,expr2)}
end

rule :notExpr do
  match(:compExpr)
```

```
  match(not_regexp, :notExpr) {| _, not_expr|
    LogicExprNode.new(not_expr, 'not')}
end

rule :xorExpr do
  match(:andExpr)
  match(:xorExpr, xor_regexp, :andExpr) {|xor_expr, _, and_expr|
    LogicExprNode.new(xor_expr,'xor', and_expr)}
end

rule :andExpr do
  match(:notExpr)
  match(:andExpr, and_regexp, :notExpr) {|and_expr, _, not_expr|
    LogicExprNode.new(and_expr,'and', not_expr)}
end

rule :orExpr do
  match(:xorExpr)
  match(:orExpr, or_regexp, :xorExpr) {|or_expr, _, xor_expr|
    LogicExprNode.new(or_expr,'or', xor_expr)}
end

rule :conditional do
  match(:orExpr)
  match('(', :orExpr, ')') {|_, or_expr, _|
    LogicExprNode.new(or_expr)}
end

rule :addiExpr do
  match(:multiExpr)
  match(:addiExpr, '+', :multiExpr) {|a_expr, op, m_expr|
    AritmExprNode.new(a_expr,op,m_expr)}
  match(:addiExpr, '-', :multiExpr) {|a_expr, op, m_expr|
    AritmExprNode.new(a_expr,op,m_expr)}
end

rule :multiExpr do
  match(:multiExpr, '*', :multiExpr) {|m_expr1, op, m_expr2|
    AritmExprNode.new(m_expr1,op,m_expr2)}
  match(:multiExpr, '/', :multiExpr) {|m_expr1, op, m_expr2|
    AritmExprNode.new(m_expr1,op,m_expr2)}
  match(:multiExpr, '%', :multiExpr) {|m_expr1, op, m_expr2|
    AritmExprNode.new(m_expr1,op,m_expr2)}
  match(:negate)
end

rule :num do
  match(:assign)
  match(:funcCall)
  match(:var)
  match(:globalVar)
  match(:numbers)
  match(:paranthesis)
end

rule :negate do
```

```
  match(:num)
  match('-', :num) {|_, num| NegateNode.new(num)}
end

rule :paranthesis do
  match('(', :addiExpr, ')') {|_, a_expr, _|
    AritmExprNode.new(a_expr)}
end

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

rule :forStmt do
  # the for-loop has 8 possibility, all are represented here!
  match(for_regexp, '(', :assign, ';', :conditional, ';', :assign, ')',
    '{', :stmts, '}'
    ) {|_, _, assign1, _, cond, _, assign2, _, _, stmts, _|
    ForNode.new(assign1,cond,assign2,stmts)}
  match(for_regexp, '(', :assign, ';', :conditional, ';', :assign, ')',
    '{', :stmts, '}'
    ) {|_, _, assign1, _, cond, _, assign2, _, _, stmts, _|
    ForNode.new(assign1,cond,assign2,stmts)}
  match(for_regexp, '(', ';', ';', ')', '{', :stmts, '}'
    ) {|_, _, _, _, _, _, stmts, _|
    ForNode.new(nil,nil,nil,stmts)}
  match(for_regexp, '(', :assign, ';', ';', ')', '{', :stmts, '}'
    ) {|_, _, assign1, _, _, _, _, stmts, _|
    ForNode.new(assign1,nil,nil,stmts)}
  match(for_regexp, '(', :assign, ';', :conditional, ';', ')',
    '{', :stmts, '}') {|_, _, assign1, _, cond, _, _, _, stmts, _|
    ForNode.new(assign1,cond,nil,stmts)}
  match(for_regexp, '(', :assign, ';', ';', :assign, ')',
    '{', :stmts, '}' ) {|_, _, assign1, _, _, assign2, _, _, stmts, _|
    ForNode.new(assign1,nil,assign2,stmts)}
  match(for_regexp, '(', ';', :conditional, ';', ')', '{', :stmts, '}'
    ) {|_, _, _, cond, _, _, _, stmts, _|
    ForNode.new(nil,cond,nil,stmts)}
  match(for_regexp, '(', ';', :conditional, ';', :assign, ')','{',
    :stmts, '}') {|_, _, _, cond, _, assign2, _, _, stmts, _|
    ForNode.new(nil,cond,assign2,stmts)}
  match(for_regexp, '(', ';', ';', :assign, ')','{', :stmts, '}'
    ) {|_, _, _, _, assign2, _, _, stmts, _|
    ForNode.new(nil,nil,assign2,stmts)}
end

rule :ifStmt do
  match(if_regexp, '(', :conditional, ')', '{', :stmts, '}', :else
       ) {|_, _, cond, _,  _, stmts, _, _else|
```

```
        IfNode.new(cond,stmts,_else)}
  match(if_regexp, '(', :conditional, ')', '{', :stmts, '}'
        ) {|_, _, cond, _,  _, stmts, _| IfNode.new(cond,stmts)}
end

rule :else do
  match(elseif_regexp, '(', :conditional, ')', '{', :stmts, '}', :else
        ) {|_, _, cond, _, _, stmts, _, _else|
        IfNode.new(cond,stmts,_else)}
  match(elseif_regexp, '(', :conditional, ')', '{', :stmts, '}'
        ) {|_, _, cond, _, _, stmts, _| IfNode.new(cond,stmts)}
  match(else_regexp, '{', :stmts, '}'
        ) {|_, _, stmts, _| IfNode.new(true,stmts)}
end


rule :whileStmt do
  match(while_regexp, '(', :conditional, ')', '{', :stmts, '}'
    ) {|_, _, cond, _, _, stmts, _| WhileNode.new(cond,stmts)}
end

rule :funcDef do
  match('def', :identifier, '(', :funcDefParamList, ')', '{',  :stmts,
        '}') {|_, id, _, list, _, _, stmts, _|
    FuncDefNode.new(id,stmts,list)}
  match('def', :identifier, '(', ')', '{', :stmts, '}'
        ) {|_, id, _, _, _, stmts, _| FuncDefNode.new(id,stmts)}
end

rule :funcDefParamList do
  match(:funcDefParamDef, ',', :funcDefParamList) {|_def, _, list|
        FuncDefParamListNode.new(_def,list)}
  match(:funcDefParamDef) {|_def| FuncDefParamListNode.new(_def)}
end

rule :funcDefParamDef do
  match(:identifier) {|id| FuncDefParamNode.new(id)}
end

rule :funcCall do
  match(:identifier, '(', :funcCallArgList, ')') {|id, _, list, _|
    FuncCallNode.new(id,list)}
  match(:identifier, '(', ')') {|id, _, _| FuncCallNode.new(id)}
end

rule :funcCallArgList do
  match(:funcCallArgDef, ',', :funcCallArgList) {|_def, _, list|
    FuncCallArgListNode.new(_def,list)}
  match(:funcCallArgDef) {|_def| FuncCallArgListNode.new(_def)}
end

rule :funcCallArgDef do
  match(:expr) {|expr| FuncCallArgNode.new(expr)}
end

rule :numbers do
```

```
  match(Integer) {|int| IntegerNode.new(int)} # also matches bignum
  match(Float) {|float| FloatNode.new(float)}
end

rule :string do
  match(string_regexp) {|string| StringNode.new(eval(string))}
end

rule :vars do
  match(:var, ',', :vars) {|var, _,  vars| VarsNode.new(var,vars)}
  match(:globalVar, ',', :vars) {|global, _,  vars|
    VarsNode.new(global,vars)}
  match(:var) {|var| VarsNode.new(var)}
  match(:globalVar) {|global| VarsNode.new(global)}
end

rule :assign do
  match(:vars, '=', :funcCall) {|vars, _,  call|
    AssignReturnNode.new(vars,call)}
  match(:globalVar, '++') {|global, inc| AssignNode.new(global,inc)}
  match(:globalVar, '++') {|global, inc| AssignNode.new(global,inc)}
  match(:var, '++') {|var, inc| AssignNode.new(var,inc)}
  match(:globalVar, '--') {|global, dec| AssignNode.new(global,dec)}
  match(:var, '--') {|var, dec| AssignNode.new(var,dec)}
  match(:globalVar, '=', :expr) {|global, _, expr|
    AssignNode.new(global,expr)}
  match(:var, '=', :expr) {|var, _, expr| AssignNode.new(var,expr)}
end

rule :term do
  match(:input) {|input| TermNode.new(input)}
  match(:boolean) {|bool| TermNode.new(bool)}
  match(:string) {|str| TermNode.new(str)}
  match(:array) {|array| TermNode.new(array)}
  match(:hash) {|hash| TermNode.new(hash)}
end

rule :array do
  match(Array) {|array| ArrayNode.new(array)}
end

rule :hash do
  match(Hash) {|hash| HashNode.new(hash)}
end

rule :boolean do
  match('true') {BoolNode.new(true)}
  match('false') {BoolNode.new(false)}
end

rule :var do
  match(:identifier) {|id| VarNode.new(id)}
end

rule :globalVar do
  match(globalvar_regexp) {|global|
```

```
        GlobalVarNode.new(IdentifierNode.new(global))}
    end

    rule :identifier do
      match(var_regexp) {|id| IdentifierNode.new(id)}
    end
  end
end

def OpenJ.get_return
  if (@@func != nil and @@func_value != nil)
    return [@@func, @@func_value]
  else
    return [nil, nil]
  end
end

def OpenJ.set_return(func=nil,value=nil)
  if(func != nil)
    @@func = func

  elsif (func == nil and value != nil)
    @@func_value = value
  end
end

def OpenJ.init
  @@scope = Variables.new(nil, "top")
  @@functions = {}
end

def OpenJ.get_func(funcname)
  if(@@functions.has_key?(funcname))
    return [funcname, @@functions[funcname]["paramlist"],
          @@functions[funcname]["suite"],
          @@functions[funcname]["scope"]]
  else
    return [nil, nil, nil, nil]
  end
end

def OpenJ.set_func(funcname, paramlist, suite, scope=nil)
  @@functions[funcname] = {"paramlist"=>paramlist,
    "suite"=>suite, "scope"=>scope}
end

def OpenJ.set_func_scope(funcname,scope)
  @@functions[funcname]["scope"] = scope
end

def OpenJ.set_scope(scope)
  @@scope = scope
end

def OpenJ.get_scope()
  return @@scope
```

```
  end


  def create_keyword_list()
    @keywords = []
    @keywords << "[Ff]or"
    @keywords << "[Ww]hile"
    @keywords << "[Ii]f"
    @keywords << "[Ee]lse[Ii]f"
    @keywords << "[Ee]lse"
    @keywords << "[Aa]nd"
    @keywords << "[Oo]r"
    @keywords << "[Xx]or"
    @keywords << "[Nn]ot"
    @keywords << "break"
    @keywords << "def"
    @keywords << "input"
    @keywords << "return"
    @keywords << "print"
    @keywords << "true"
    @keywords << "false"
  end

  def create_regexp(exp,keywords=nil)
    if(keywords != nil)
      words = "^(((?!("
      keywords.each do |keyword|
        words += "^" + keyword + "$|"
      end
      words = words.chop
      words += "))" + exp
      words += "))$"
    else
      words = exp
    end
    return Regexp.new(words)
  end

  def run(filename ="")
      str = File.open(filename).read
      return @diceParser.parse(str)
  end

  def log(state = false)
    if state
      @diceParser.logger.level = Logger::DEBUG
    else
      @diceParser.logger.level = Logger::WARN
    end
  end
end


class Variables
  def initialize(next_level=nil, id=nil)
```

```
  @next_level = next_level
  @id = id
  @vars = {}
end

def get_var(name)
  if(@vars.has_key?(name))
    return @vars[name]
  else
    if(@id == 'top' and not name.match(/^@\w+/))
      name = "@" + name
      if(@vars.has_key?(name))
        return @vars[name]
      end
    end
    if(@next_level != nil)
      @next_level.get_var(name)
    else
      return nil
    end
  end
end

def set_var(name, value)
  if(name.class == Array and value.class == Array)
    index = 0
    while(index < name.length) do
      if(name[index].to_s.match(/^@\w+/))
        if(@id == "top")
          add_var(name[index], value[index])
        else
          if(@next_level != nil)
            @next_level.set_var(name[index], value[index])
          end
        end
      else
        add_var(name[index], value[index])
      end
      index += 1
    end
  else
    if(name.match(/^@\w+/))
      if(@id == "top")
        add_var(name, value)
      else
        if(@next_level != nil)
          @next_level.set_var(name, value)
        end
      end
    else
      add_var(name, value)
    end
  end
end

def add_var(name, value)
```

```
    @vars[name] = value
  end

  private :add_var
end

def readfile(filename="")
  if(File.exist?(filename))
    OpenJ.new.run(filename)
  else
    puts "'" + filename + "' doesnot exist!"
    puts "OpenJ terminated!"
  end
end

readfile(ARGV[0])
```

5.2 rdparse.rb

```
#!/usr/bin/env ruby

# 2010-02-11 New version of this file for the 2010 instance of TDP007
#   which handles false return values during parsing, and has an easy way
#   of turning on and off debug messages.

require 'logger'

class Rule

  # A rule is created through the rule method of the Parser class, like this:
  #   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
  # @logger
  def initialize(name, parser)
    # @logger = parser.logger
    # The name of the expressions this rule matches
    @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
    @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?
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result if result.nil?
      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 e.g.
      #   [: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
          if result.last.nil?
            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, :logger

  class ParseError < RuntimeError
  end

  def initialize(language_name, &block)
    # @logger = Logger.new(STDOUT)
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name = language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @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")
          # 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
end

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

  # 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
```

5.3 Nodes.rb

```
#-----------------------------------[A]-----------------------------------

class AritmExprNode

  def initialize(expr1, aritmop=nil, expr2=nil)
    @expr1 = expr1
    @aritmoperator = aritmop
    @expr2 = expr2
  end

  def eval
    if (@aritmoperator == nil)
      return @expr1.eval()
    end
    expr1 = @expr1.eval()
    expr2 = @expr2.eval()
    if(expr1.class == ReturnNode)
      expr1 = expr1.get_value()[0]
    end
    if(expr2.class == ReturnNode)
      expr2 = expr2.get_value()[0]
    end
    if(expr1 == nil or expr1.class == NullNode)
      expr1 = 0
    end
    if(expr2 == nil or expr2.class == NullNode)
      expr2 = 0
    end
```

```
    if(@aritmoperator == "/" and (expr1 < expr2))
      expr1 = expr1.to_f
    end
    Kernel.eval("#{expr1} #{@aritmoperator} #{expr2}")
  end
end

class ArrayNode

  def initialize(array)
    @array = array
  end

  def eval
    return @array
  end
end

class AssignNode

  def initialize(var, term)
    @var = var
    @term = term
  end

  def eval
    @scope = OpenJ.get_scope()
    var = @scope.get_var(@var.get_name())
    if(@term == "--")
      if(var != nil)
        if(var.class == Fixnum)
          var -= 1
        end
        @scope.set_var(@var.eval(), var)
      end
    elsif(@term == "++")
      if(var != nil)
        if(var.class == Fixnum)
          var += 1
        end
        @scope.set_var(@var.get_name(), var)
      end
    else
      term = @term.eval()
      @scope.set_var(@var.get_name(), term)
      return term
    end
  end
end

class AssignReturnNode

  def initialize(vars, funccall)
    @vars = vars
    @funccall = funccall
  end
```

```
  def eval
    @scope = OpenJ.get_scope()
    vars_list = @vars.eval()
    return_object = @funccall.eval()
    if(return_object.class == ReturnNode)
      vars_length = vars_list.length
      return_value = return_object.get_value()
      return_length = return_value.length
      if(vars_length > return_length)
        while(return_value.length <= vars_length) do
          return_value << NullNode.new()
        end
      elsif(vars_length < return_length)
        while(vars_list.length <= return_length) do
          vars_list.pop
        end
      end
      @scope.set_var(vars_list, return_value)
    end
    return
  end
end

#-------------------------------------[B]-------------------------------------

class BoolNode

  def initialize(bool)
    @bool = bool
  end

  def eval
    return @bool
  end
end

class BreakNode

  def initialize
  end

  def eval
    return self
  end
end

#-------------------------------------[C]-------------------------------------

class CompExprNode

  def initialize(expr1, compop=nil, expr2=nil)
    @expr1 = expr1
    @compoperator = compop
    @expr2 = expr2
  end
```

```
  def eval
    if (@compoperator == nil)
      return @expr1.eval()
    end

    if(@compoperator == "<>")
      @compopertator = "!="
    end
    return Kernel.eval("#{@expr1.eval()} #{@compoperator} #{@expr2.eval()}")
  end
end

#-------------------------------------[D]-------------------------------------
#-------------------------------------[E]-------------------------------------
#-------------------------------------[F]-------------------------------------

class FloatNode

  def initialize(float)
    @float = float
  end

  def eval
    return @float
  end
end

class ForNode
  def initialize(var=nil, condition=nil, varchange=nil, suite=nil)
    @var = var
    @condition = condition
    @varchange = varchange
    @suite = suite
  end

  def eval
    old_scope = OpenJ.get_scope()
    @scope = OpenJ.set_scope(Variables.new(old_scope))
    if(@var != nil)
      @var.eval()
    else
      @var = nil
    end
    if(@condition == nil)
      @condition = BoolNode.new("true")
    end
    while(@condition.eval()) do
      if(@suite != nil)
        return_value = @suite.eval()
      end
      if(return_value.class == ReturnNode)
        break
      elsif(return_value.class == BreakNode)
        return_value = NullNode.new()
        break
      end
```

```
      if(@varchange != nil)
        @varchange.eval()
      end
    end
    OpenJ.set_scope(old_scope)
    return return_value
  end
end

class FuncCallNode

  def initialize(name, arglist=nil)
    @name = name
    @arglist = arglist
  end

  def eval
    @scope = OpenJ.get_scope()
    if(@name != "main")
      name = @name.eval()
    else
      name = @name # only used by the ProgramNode's start functioncall for "main"
    end
    funcname, paramlist, suite, scope = OpenJ.get_func(name)
    if(scope == nil)
      scope = Variables.new(@scope)
    end
    OpenJ.set_func_scope(funcname,scope)
    if(@arglist == nil)
      arglist = []
    else
      arglist = @arglist.eval()
    end
    if(funcname != nil and scope != nil)
      if(paramlist.length == arglist.length)
        OpenJ.set_scope(scope)
        scope.set_var(paramlist, arglist)
        return_value = suite.eval()
        OpenJ.set_scope(@scope)
      else
        puts "Error: Number of given value doesn't match number of arguments
required!"
      end
    else
      puts "'" + name + "'"
      puts "Error: Called function is not defined!"
    end
    return return_value
  end
end

class FuncCallArgListNode

  def initialize(arg, args=nil)
    @arg = arg
    @args = args
```

```
  end

  def eval
    return_value = @arg.eval()
    if(@args != nil)
      return_value += @args.eval()
    end
    return return_value
  end
end

class FuncCallArgNode

  def initialize(arg)
    @argument = arg
  end

  def eval()
    return_value = @argument.eval()
    if (return_value.class == ReturnNode)
      return_value = return_value.get_value()[0]
    end
    return [return_value]
  end
end

class FuncDefNode

  def initialize(name, suite=nil, paramlist=nil)
    @name = name
    if (paramlist == nil)
      @parameterlist = []
    else
      @parameterlist = paramlist.eval()
    end
    @suite = suite
    if(@name != nil)
      OpenJ.set_func(@name.eval(), @parameterlist, @suite)
    else

    end
  end

  def eval
  end
end

class FuncDefsNode

  def initialize(funcdef, funcdefs=nil)
    @funcdef = funcdef
    @funcdefs = funcdefs
  end

  def eval
  end
end
```

```
class FuncDefParamNode

  def initialize(parameter)
    @parameter = parameter
  end

  def eval
    return [@parameter.eval()]
   end
end

class FuncDefParamListNode

  def initialize(param, params=nil)
    @param = param
    @params = params
  end

  def eval
    return_value = @param.eval()
    if(@params != nil)
      return_value += @params.eval()
    end
    return return_value
  end
end

#-------------------------------------[G]-------------------------------------

class GlobalVarNode

  def initialize(name)
    @name = name
  end

  def get_name
    return @name.eval()
  end

  def eval
    @scope = OpenJ.get_scope()
    return_value = @scope.get_var(@name.id)
    if(return_value == nil)
       return NullNode.new()
    end
    return return_value
  end
end

#-------------------------------------[H]-------------------------------------

class HashNode

  def initialize(hash)
    @hash = hash
  end
```

```
  def eval
    return @hash
  end
end

#-----------------------------------[I]-----------------------------------

class IdentifierNode

  def initialize(id)
    @id = id
  end

  def eval
    return @id
  end
end

class IfNode

  def initialize(conditional, ifbranch, elsebranch=nil)
    @conditional = conditional
    @ifbranch = ifbranch
    @elsebranch = elsebranch
  end

  def eval
    if(@conditional == true)
      conditional = true
    else
      conditional = @conditional.eval()
    end
    if (conditional)
      return_value = @ifbranch.eval()
    else
      if (@elsebranch != nil)
        return_value = @elsebranch.eval()
      end
    end
    return return_value
  end
end

class InputNode

  def initialize(var)
    @var = var
  end

  def eval
    in_value = STDIN.gets
    begin
      return_value = Integer(in_value)
    rescue ArgumentError
      return_value = in_value
    end
    if(return_value.class != Fixnum)
```

```
      begin
        return_value = Float(in_value)
      rescue ArgumentError
        return_value = in_value
      end
    end
    return return_value
  end
end

class IntegerNode

  def initialize(int)
    @int= int
  end

  def eval
    return @int
  end
end

#------------------------------------[J]-----------------------------------
#------------------------------------[K]-----------------------------------
#------------------------------------[L]-----------------------------------

class LogicExprNode

  def initialize(expr1, logicop=nil, expr2=nil)
    @expr1 = expr1
    @logicoperator = logicop
    @expr2 = expr2
  end

  def eval
    if (@logicoperator == nil)
      return @expr1.eval()
    elsif(@logicoperator == "xor")
      @logicoperator = "^"
    elsif(@logicoperator == "not")
      return Kernel.eval("#{@logicoperator} #{@expr1.eval()}")
    end
    return Kernel.eval("#{@expr1.eval()} #{@logicoperator} #{@expr2.eval()}")
  end
end

#------------------------------------[M]-----------------------------------
#------------------------------------[N]-----------------------------------

class NegateNode

  def initialize(expr)
    @expr = expr
  end

  def eval
    return -(@expr.eval())
```

```
  end
end

class NullNode

  def initialize
    @value = nil
  end

  def eval
  end
end

#-------------------------------------[O]-------------------------------------
#-------------------------------------[P]-------------------------------------

class PrintNode

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

  def eval
    if(@expr != nil)
      print_value = @expr.eval()
      if(print_value.class == ReturnNode)
        print_value = print_value.get_value()
      end
      if(print_value.class == NullNode)
        puts "nil"
      else
        puts print_value
      end
    else
      puts
    end
  end
end

class ProgramNode

  def initialize(program)
    @program = program
    main = FuncCallNode.new('main')
    main.eval() # starts the main function
    return nil
  end
end

#-------------------------------------[Q]-------------------------------------
#-------------------------------------[R]-------------------------------------

class ReturnExprNode

  def initialize(expr)
    @expr = expr
```

```
    @return_value = NullNode.new()
  end

  def eval()
    return_value = [@expr.eval()]
    return return_value
  end
end

class ReturnExprsNode

  def initialize(expr, exprs=nil)
    @expr = expr
    @exprs = exprs
  end

  def eval
    return_value = @expr.eval()
    if(@exprs != nil)
      return_value += @exprs.eval()
    end
    return return_value
  end
end

class ReturnNode

  def initialize(returnExprs=nil)
    @returnExprs = returnExprs
    @return_value = NullNode.new()
  end

  def get_value
    return @return_value
  end

  def eval
    if(@returnExprs != nil)
      @return_value = @returnExprs.eval()
    else
      @return_value = []
    end
    return self
  end
end

#-------------------------------------[S]-------------------------------------

class StmtsNode

  def initialize(stmt, stmts=nil)
    @stmt = stmt
    @stmts = stmts
  end

  def eval
    return_value = @stmt.eval()
```

```
    if (return_value.class == ReturnNode or return_value.class == BreakNode)
      return return_value
    end
    if(@stmts != nil)
      return_value = @stmts.eval()
    end
    return return_value
  end
end

class StringNode

  def initialize(string)
    @string = string
  end

  def eval
    return @string
  end
end

#-------------------------------------[T]-------------------------------------

class TermNode

  def initialize(term)
    @term = term
  end

  def eval
    return @term.eval()
  end
end

#-------------------------------------[U]-------------------------------------
#-------------------------------------[V]-------------------------------------

class VarNode

  def initialize(name)
    @name = name
  end

  def get_name
    return @name.eval()
  end

  def eval
    @scope = OpenJ.get_scope()
    return_value = @scope.get_var(@name.eval())
    if(return_value == nil)
      return NullNode.new()
    end
    return return_value
  end
end
```

```
class VarsNode

  def initialize(var, vars=nil)
    @var = var
    @vars = vars
  end

  def eval
    return_value = [@var.get_name()]
    if(@vars != nil)
      return_value += @vars.eval()
    end
    return return_value
  end
end

#-------------------------------------[W]-------------------------------------

class WhileNode

  def initialize(expr, suite=nil)
    @expr = expr
    @suite = suite
  end

  def eval
    if (@suite != nil)
      while(@expr.eval()) do
        return_value = @suite.eval()
        if(return_value.class == ReturnNode)
          break
        elsif(return_value.class == BreakNode)
          return_value = NullNode.new()
          break
        end
      end
    end
    return return_value
  end
end

#-------------------------------------[X]-------------------------------------
#-------------------------------------[Y]-------------------------------------
#-------------------------------------[Z]-------------------------------------
```

Bilaga A. Grammatik

Teckenförklaring:
Allt innanför hakparenteser ”[]” är valfritt,
? betyder noll eller en gång
* betyder noll eller fler gånger
+ betyder en eller flera

<**program**> ::= <**funcDefs**>

<**funcDefs**> ::= <**funcDef**> [<**funcDefs**>]?

<**stmts**> ::= <**stmt**> [<**stmts**>]?

<**stmt**> :== <**break**>
| <**special**>
| <**return**>
| <**forStmt**>
| <**ifStmt**>
| <**whileStmt**>
| <**funcDef**>
| <**funcCall**>
| <**assignReturn**>

<**break**> ::= 'break'

<**return**> ::= 'return' [<**returnExprs**>]?

<**returnExprs**> ::= <**returnExpr**> [',' <**returnExprs**>]?

<**returnExpr**> ::= <**expr**>

<**special**> ::= <**print**>

<**input**> ::= 'input'

<**print**> ::= 'print' [<:**expr**>]?

<**expr**> ::= <**addiExpr**>
| <**term**>

<**compExpr**> ::= <**addiExpr**>
| <**boolean**>
| <**compExpr**> <**compOperator**> <**compExpr**>

<**notExpr**> ::= <**compExpr**>
| ('Not' | 'Not') <**notExpr**>

<**xorExpr**> ::= <**andExpr**>
| <**xorExpr**> ('Xor' | 'xor') <**andExpr**>

<**andExpr**> ::= <**notExpr**>
| <**andExpr**> ('And' | 'and) <**notExpr**>

<**orExpr**> ::= <**xorExpr**>
| <**orExpr**> (' Or' | 'or') <**xorExpr**>

<**conditional**> ::= <**orExpr**>
| '(' <**orExpr**> ')'

<**addiExpr**> ::= <**multiExpr**>
| <**addiExpr**> '+' <**multiExpr**>
| <**addiExpr**> '-' <**multiExpr**>

<**multiExpr**> ::= <**multiExpr**> '*' <**multiExpr**>
| <**multiExpr**> '/' <**multiExpr**>
| <**multiExpr**> '%' <**multiExpr**>
| <**negate**>

<**negate**> ::= <**num**>
| '-' <**num**>

<**num**> ::= <**assign**>
| <**funcCall**>
| <**var**>
| <**globalVar**>
| <**numbers**>
| <**paranthesis**>

<**paranthesis**> ::= '(' <**addiExpr**> ')'

<**compOperator**> ::= '<'
| '>'
| '=='
| '>='
| '<='
| '<>'
| '!='

<**forStmt**> ::= ('For' | 'for') '(' [<**assign**>]? ';' [<**conditional**>]? ';' [<**assign**>]? ')'
'{' <**stmts**> '}'

<**ifStmt**> ::= ('If' | 'if') '(' <**conditional**> ')' '{' <**stmts**> '}' [<**else**>]?

<**else**> ::= ('Elseif' | 'ElseIf' | 'elseIf' | 'elseif') '(' <**conditional**> ')' '{' <**stmts**> '}' [<**else**>]?
| ('Else' | 'else') '{' <**stmts**> '}'

<**whileStmt**> ::= ('While' | 'while') '(' <**conditional**> ')' '{' <**stmts**> '}'

<**funcDef**> ::= 'def' <**identifier**> '(' [<**funcDefParamList**>]? ')' '{' <**stmts**> '}'

<**funcDefParamList**> ::= <**funcDefParamDef**> ',' [<**funcDefParamList**>]?

<**funcDefParamDef**> ::= <**identifier**>

<**funcCall**> ::= <**identifier**> '(' [<**funcCallArgList**>]? ')'

<**funcCallArgList**> ::= <**funcCallArgDef**> [',' <**funcCallArgList**>]?

<**funcCallArgDef**> ::= <**funcCall**>
| <**assign**>
| <**expr**>

<**numbers**> ::= <**Integer**> (Använder den inbyggda datatypen Integer i Ruby)
| <**Float**> (Använder den inbyggda datatypen Float i Ruby)

<**string**> ::= (('.*") | ('.*')) (Matchar alla tecken som finns innanför ” ” och ' ')

<**assignReturn**> ::= <**vars**> '=' <**funcCall**>
| <**assign**>

<**vars**> ::= <**var**> [',' <**vars**>]?
| <**globalVar**> [',' <**vars**>]?

<**assign**> ::= <**vars**> '=' <**funcCall**>
| <**globalVar**> ('++'|'--')
| <**var**> ('++'|'--')
| <**globalVar**> '=' <**expr**>
| <**var**> '=' <**expr**>

<**term**> ::= <**input**>
| <**boolean**>
| <**string**>
| <**array**>
| <**hash**>

<**array**> ::= Array (Använder den inbyggda datatypen Array i Ruby)

<**hash**> ::= Hash (Använder den inbyggda datatypen Hash i Ruby)

<**boolean**> ::= ('true' | 'false')

<**var**> ::= <**identifier**>

<**globalVar**> ::= "@"[_a-zA-Z]+[_a-zA-Z0-9]*

<**identifier**> ::= [_a-zA-Z]+[_a-zA-Z0-9]*