Linköpings universitet
Innovativ Programmering
TDP019 Projekt: Datorspråk

flip/flop

ett helt flippat språk

TDP019 - Projekt: Datorspråk
Vårterminen 2012

Johan Wänglöf johwa457@student.liu.se
Henrik Forsberg henfo939@student.liu.se

Sammanfattning

Målet med TDP019 är att vi ska lära oss hur man gör sitt eget programmeringsspråk med hjälp av en parser skriven i Ruby. Första halvan av vårterminen gjorde vi färdigt vår grammatik, som vi sedan implementerade andra halvan av terminen.

Flip/flop har många av de traditionella syntaxer som de flesta språken har fast med en liten twist, syntaxen är “flippad” vilket bland annat betyder att villkorssatserna kommer i slutet av blocken istället för i början. Exempel på detta hittar du under Användarhandledning i detta dokument.

Språket är interpreterat och använder RDparser som i sin tur är skrivet i programmeringsspråket Ruby.

1. Inledning

Vi fick i uppgift att skriva vårt eget programmeringsspråk i kursen *TDP019 Projekt: Datorspråk* under andra terminen av vårt första år i Innovativ Programmering på Linköpings universitet.

1.1 Beskrivning av språket flip/flop

Vi kom fram till att det skulle vara roligt att göra ett språk som är bakvänt jämfört med några av de populäraste språk som används idag. Det finns inget direkt användningsområde där *flip/flop* utmärker sig extra väl utan det är bara en rolig idé som vi tyckte skulle vara intressant att se om det skulle fungera eller inte. Eftersom det inte finns något speciellt användningsområde för språket har vi inte någon specifik målgrupp vi skrivit språket för.

1.2 Syfte

Syftet med denna kurs var att lära oss hur ett programmeringsspråk fungerar i grunden. Hur kompilatorer förstår alla syntaxer man använder för att skapa ett program.

1.3 Idé

Vår idé är mycket enkel, att skriva ett programmeringsspråk som sätter villkorssatserna i slutet av ett block och därmed göra ett rätt så unikt språk som inte är så lätt att sätta sig in i.

2. Användarhandledning

Under hela denna handledning kommer vi anta att användaren använder någon distribution av Linux.
Denna del är till för den nya användaren av *flip/flop*. Här får denne hjälp att komma igång med installationen av språket samt hur man skriver språkets syntax.
Handledningen kommer börja med hur man kommer igång att koda med språket, alltså vad man behöver ha installerat på sin dator för det ska fungera korrekt. Efter detta visas exempel på vår syntax och därefter några kodexempel i *flip/flop*.

2.1 Komma igång/Installation

Det allra första som du behöver är att ha Ruby installerat.

- Om du använder Linux är det lättast att hämta det via din pakethanterare.
- På http://www.ruby-lang.org/en/downloads/ finns det fler installationsinstruktioner.

2.2 Köra flip/flop-kod

2.2.1 Installation

Flip/flop levereras i ett ziparkiv. I detta arkiv finns filerna för att kunna starta språket, samt en mapp med dokumentation och några testfiler. Om man inte vill ladda hem ett ziparkiv med språket kan man klona Johans git arkiv från URL:n *git://github.com/jwanglof/TDP019.git*
Observera att detta kommer enbart vara en “read-only” kopia och du kan inte ändra på källkoden.
Mer information om hur git fungerar kan du läsa från github:s hjälphemsida på följande URL: *http://help.github.com/*

2.2.2 Testa språket med hjälp av interpretatorn

Det absolut lättaste att komma igång och koda är att använda språkets interpretator som finns inbyggd i *flip/flop*. För att börja använda språket, gör följande:

- Gå in i Rubys interpretator (öppna upp ett nytt terminalfönster och skriv *irb*)
- Skriv *load 'run.rb'*
- Du bör nu ha en ny rad som ser ut som *f/f > .*
- Du kan nu prova språket.

OBS. Det är ej rekommenderat att testa if-satser, for- eller while-loopar i interpretatorn då det blir väldigt rörigt när man skriver detta på en rad.

2.2.3 Testa språket med hjälp av en extern fil

Det andra alternativet är att skapa en fil med *flip/flop*-kod. För att köra denna fil skriver man bara:
ruby flipflop_source.rb filnamnet när man står i mappen där källkoden ligger.
Vi har bifogat några exempelfiler som ligger i mappen *test* och för att testa någon av dessa filer skriver användaren:
ruby flipflop_source.rb test/filnamn

2.3 Programmera flip/flop-kod

2.3.1 Datatyper

Eftersom språket var mest gjort för att lära oss hur ett språk verkligen fungerar bestämde vi oss att enbart implementera 3st olika datatyper. De som fungerar är:

- Flyttal
- Heltal
- Strängar

2.3.2 Variabler och tilldelning

En variabel kan börja med någon bokstav mellan a-z (både stora och små bokstäver fungerar), med en siffra mellan 0-9 eller med ett "underscore". Ett variabelnamn måste vara åtminstone en symbol som vi definerade innan men efter det kan det vara godtyckligt många symboler.

Vi antar att de flesta som läser denna rapport har programmerat innan och vet hur man deklarerar en variabel i de flesta språk, nämligen:

```
varName = "En fin variabel"
```

Men eftersom *flip/flop* är som namnet antyder "flippat" deklarerar man en variabel såhär i *flip/flop*:

```
“En fin variabel” = varName
```

Nu antar du säkert vad vi menar med “flippat”. Det är lite krångligt att börja tänka omvänt när man ska skriva och det tog tid för oss också att vänja oss vid det.

2.3.3 Operatorer och operationer

För att ett språk ska kunna användas till någonting måste användaren kunna lägga ihop tal för att räkna ut något. Eller kanske för att se om ett uttryck är sant eller falskt. För att göra detta möjligt i detta språk har vi delat upp dem i aritmetiska-, logiska- och relations-operander.

- Aritmetiska operander: +, -, *, / och %
- Logiska operander: || och &&
- Relationsoperander: <, <=, >, >=, == och !=

Dessa kan användas för att se t.ex. om två tal är desamma, se om två strängar har samma värde eller addera två tal och se om det blir en specifik summa. Eller kan man använda språket som en kalkylator.

- Aritmetiska uttryck

```
2 + 2 scream                              #Skriver ut 4
2 + 2 * 6 scream                          #Skriver ut 14
2 - 2 + 4 + 4 scream                      #Skriver ut 8
5 % 2 scream                              #Skriver ut 1
```

- Logiska uttryck

```
5 + 5 == 10 && 2 + 2 == 4 scream          #true
5 + 5 == 11 || 2 + 2 == 4 scream          #true
```

- Relationsuttryck

```
5 > 10 scream                             #false
5 > 1 scream                              #true
```

2.3.4 Repetitionssatser

Vi bestämde oss att vi vårt språk heter for- och while-loopar samma sak, nämligen “loop”. De fungerar som de gör i andra språk, fast det är självklart en “flipp” här också.

2.3.4.1 For-loopar

Här är ett enkelt exempel på en for-loop som loopar ut en variabel som har ett heltalsvärde:

```
pool
      i scream
loop (1 = i; 5 < i; i++)
```

Detta kommer resultera i:

```
2
3
4
5
6
```

Som syns är allt “flippat” inom villkorssatserna och likadant i syntaxen

2.3.4.2 While-loopar

Ett enkelt exempel på en while-loop som kommer ha samma resultat som for-loopen ovanför:

```
1 = i
pool
      i++
      i scream
loop (i < 6)
```

2.3.5 Vilkorssatser

En if-sats i *flip/flop* består utav åtminstone en del av uttrycket, men man kan bygga på med fler om man känner för det.
Den obligatoriska delen av en if-sats är “fi (uttryck)” i vårt språk. De delar man kan bygga på är “fi esle (uttryck)” och “esle”

2.3.5.1 If med else

```
fi (2 < 1)
      “Villkoret är sant” scream
esle
      ”Villkoret är falskt” scream
```

2.3.5.2 If-else

```
fi (2 < 1)
      ”Villkoret är sant” scream
fi esle (2 < 3)
      ”Villkoret är sant” scream
```

2.3.5.3 If-else med else

```
fi (2 < 1)
      ”Villkoret är sant” scream
fi esle (2 < 3)
      ”Villkoret är sant” scream
esle
      ”Villkoren är falska” scream
else
```

2.3.6 Funktioner

Vi bestämde oss att vårt funktionsnamn börjar med *boj* och slutar med *job (argument)*. Man kan deklarera funktioner med eller utan argument och också anropa funktioner med eller inparametrar. För att anropa en funktion skriver man bara namnet på funktionen följt av inparametrar mellan två stycken parenteser.

2.3.6.1 Funktion med argument

Detta visar en funktion som adderar två tal och skriver ut detta.

```
boj
      num1 + num2 = result
      result scream
job addNumbers(num1, num2)
```

Vi har dock inte fått funktioner med argument att fungera som de ska. Det skapas en nod när man deklarerar funktionen men man kommer inte åt denna funktion via namnet efteråt.

2.3.6.2 Funktion utan argument

Samma funktion som ovan fast utan argument.

```
boj
      2 + 2 = result
      result scream
job screamFour()
```

2.3.6.3 Anrop av en funktion

Inget speciellt med detta utan visar enbart hur man kallar på funktionen i första funktionsexemplet.

```
addNumbers(2, 2)
```

Detta kommer skriva ut 4.

3. Systemdokumentation

Vi har byggt *flip/flop* i rdparsern som "parsar" igenom vår kod som vi skrivit som anger hur vår syntax ska se ut.

3.1 Mappstruktur

/
Docs/
    README
    FlipFlop.docx
test/
    cluster
    for
    function
    if
    variable
    while
run.rb
flipflop.rb[1]
node.rb[2]
rdparse.rb[3]

3.2 Lexikalisk analys

Vår lexikaliska analys består enbart utav "tokens" som vår parser använder för att matcha våra regler som vi definerat i koden. Dessa är de tokens vi har:

Dessa två tokens matchar enradskommentarer och flerradskommentarer. Om användaren använder detta syntax kommer allt efter en # eller inom ## *text* ## ignoreras av parsern.

```
token(/^#(.)*$/)
token(/^(##[\w\W\s]*##)/)
```

Detta matchar alla blanksteg (white spaces) som säger till parsern att ignorera dessa.

```
token(/^(\s)/)
```

[1] Har alla regler språket kan hantera
[2] Innehåller språkets noder
[3] Parsern vi fick som verktyg

Dessa är språkets specifika tilldelningssyntax. Du kan läsa om dessa satser under vår *Användarhandledning*.

```
token(/^(scream|yes|no|fi|esle|else|esle fi|cluster|boj|job)/) { |m| m }
```

Detta matchar språkets specifika operatorer.

```
token(/^(\++|\+|\|\*|\/|\%|\!=|\==|\=|\!|\&&|\<|\>|\<=|\>=|\(|\)|\]|\[|\|\||\
;|\,)/) { |m| m }
```

Den första och andra tar hand om negativa flyttal resp. positiva flyttal. Den tredje tar hand om alla siffror.

```
token(/^(-(\d+[.]\d+))/) {|m| m.to_f }
token(/^(\d+[.]\d+)/) {|m| m.to_f }
token(/^(\d+)/) { |m| m.to_i }
```

Dessa tar hand om strängar. Våra strängar måste börja med citationstecken.

```
token(/^('[^\']*')/) { |m| m }
token(/^("[^\"]*")/) { |m| m }
```

Denna tar hand om variablar när man deklarerar dessa.

```
token(/^([^\'\"][a-zA-Z0-9_]+[a-zA-Z0-9_]*)/) { |m| m }
```

3.3 Parsning

När man skriver ett program i *flip/flop* skapas ett abstrakt syntaxträd med noder som är objekt av olika klasser som beskriver hur denna nod ska bete sig. Den översta noden är jämt *Program* och efter denna kommer alltid en *Statement_List* som sedan innehåller de olika noderna som programsyntaxen har skapat.

3.4 Evaluering

Varje klass i språket har en evaluate-funktion som kallas då språkkonstruktionen skall exekveras. Dessa klasser som anropas är de noder som vi tidigare skrivit om. Vid exekveringen av ett program kommer trädet att traversera och de olika evalueringsfunktionerna kommer att anropas.

3.5 Scoping

För att kunna hantera variabler som ej är åtkomliga från programmets globala rymd skapar varje funktion eller loop en egen privat rymd, eller “scope”. För att lyckas med detta sparas samtliga variabler för ett scope i en hash-tabell med variabelnamnen som nycklar. Hash-tabellen sparas i sin tur sedan som ett element i en lista. På så vis blir varje element i listan ett scope och

programmet kan enkelt välja det scope som ska gälla med en enkel indexvariabel. Varje gång en funktion eller loop körs kallas en metod som räknar upp indexvariabeln och på samma sätt kallas en nedräknande metod precis innan funktionen eller loopen avslutas.

4. Reflektion

Det tog tid för oss att komma igång med implementeringen av språket då vi hade stora svårigheter med att förstå hur grammatiken skulle utformas fullt ut. Det vi fastnade på var hur aritmetiska uttryck skulle se ut rent grammatiskt, och framför allt hur prioriteringen skulle skötas. Detta gjorde att vi sköt upp implementeringen ytterligare eftersom vi kände att en grammatik med fungerande aritmetiska uttryck var viktigt att ha innan vi påbörjade den. I och med att vi kom igång så sent som vi gjorde bestämde vi oss ganska snart för att vårt språk inte skulle innehålla scopes och funktioner då vi hade hört från många av våra kamrater att dessa tagit lång tid. Efter att ha studerat publicerade språkprojekt från tidigare år och tagit hjälp av kamrater som kommit längre lyckades vi till slut få något slags grepp om huruvida vi skulle gå tillväga för att komma i mål.

Ett problem vi stötte på som tvingade oss att förändra vår syntax gällde våra if-satser. Vår syntax går ju enligt beskrivningen ovan ut på att vända på så mycket som möjligt av nyckelord och ordning av satser med mera jämfört med hur det kan se ut i andra språk. Problemet uppstod när vi, i vår if-sats, försökte evaluera villkoret i slutet av satsen istället för i början som man gör i de flesta andra språk. Här fick vi ge oss och flyttade tillbaka villkoret till början av if-satsen istället.

Sista veckan innan deadline lyckades vi på något vis få lite tid över och bestämde oss, tidigare beslut till trots, för att försöka implementera scopes och funktioner i språket. Vi lyckades inte att få till funktionerna till en början då dessa inte lyckades leverera värden från eventuella argument till funktionens parametrar vid ett funktionsanrop. Det tog några timmar med kodning och diskussion men nu fungerar detta.

Erfarenheter vi tar med oss från det här projektet är som vanligt framför allt att vi måste komma igång tidigare (även om det inte alltid är så lätt) med det som ska göras men vi tror oss även gå ur det här projektet med en mycket djupare förståelse för hur ett datorspråk är uppbyggt och fungerar, vilket väl var tanken med kursen. Känslan vi har är att vi nu fått prova på hur det kan vara att skapa ett nytt programmeringsspråk och förhoppningsvis får vi i framtiden även chansen att utforska det här området ytterligare.

5. Grammatik

```
<PROGRAM>                ::=    <STMT_LIST>

<STMT_LIST>              ::=    <STMT_LIST> <STMT>
                                | <STMT>

<STMT>                   ::=    <FUNCTION_CALL>
                                | <PRINT_STMT>
                                | <ASSIGN_STMT>
                                | <IF_STMT>
                                | <EXPRESSION>
                                | <LOOP_STMT>
                                | <FUNCTION_DECL>

<FUNCTION_CALL>          ::=    <IDENTIFIER> “(“ <ARG_LIST> “)”
                                | <IDENTIFIER> “()”

<PRINT_STMT>             ::=    <EXPRESSION> “scream”
                                | <IDENTIFIER> ”[“ <INTEGER> “] scream”

<ASSIGN_STMT>            ::=    <EXPRESSION> “=” <IDENTIFIER>

<IF_STMT>                ::=    “fi (“ <EXPRESSION> “)” <STMT_LIST> “esle”
                                <STMT_LIST> “else”
                                | “fi (“ <EXPRESSION> “)” <STMT_LIST> “esle”
                                <IF_STMT>
                                | “fi (“ <EXPRESSION> “)” <STMT_LIST>

<EXPRESSION>             ::=    <ADD_ONE>
                                | <SUBTRACT_ONE>
                                | <OR_TEST>
                                | <ARRAY>
                                | “(“ <COMPARISON> “)”
                                | <ATOM>

<ADD_ONE>                ::=    <IDENTIFIER> “++”

<SUBTRACT_ONE>           ::=    <IDENTIFIER> “--”

<OR_TEST>                ::=    <AND_TEST>
                                | <OR_TEST> “||” <AND_TEST>

<AND_TEST>               ::=    <NOT_TEST>
                                | <AND_TEST> “&&” <NOT_TEST>

<NOT_TEST>               ::=    <COMPARISON>
                                | “!” <NOT_TEST>

<EXPR_ADDITION>          ::=    <EXPR_MULTIPLICATION>
                                <EXPR_ADDITION> “+” <EXPR_MULTIPLICATION>
                                <EXPR_ADDITION> “-” <EXPR_MULTIPLICATION>
```

```
<EXPR_MULTIPLICATION>     ::=    <EXPR_UNARY>
                                 | <EXPR_MULTIPLICATION> ‘*’ <EXPR_UNARY>
                                 | <EXPR_MULTIPLICATION> ‘/’ <EXPR_UNARY>
                                 | <EXPR_MULTIPLICATION> ‘%’ <EXPR_UNARY>

<EXPR_UNARY>              ::=    <ATOM>
                                 | “-” <EXPR_UNARY>

<ARRAY>                   ::=    “cluster (“ <ARRAY_VALUES> “) =” <IDENTIFIER>

<ARRAY_VALUES>            ::=    <ATOM>
                                 | <ARRAY_VALUES> “,” <ATOM>

<COMPARISON>              ::=    <EXPR_ADDITION> <OP_RELATIONAL>
                                 <EXPR_ADDITION>
                                 | <EXPR_ADDITION>

<ATOM>                    ::=    <BOOLEAN>
                                 | <INTEGER>
                                 | <FLOAT>
                                 | <STRING>
                                 | <IDENTIFIER>

<BOOLEAN>                 ::=    “yes”
                                 | “no”

<INTEGER>                 ::=    Integer

<FLOAT>                   ::=    Float

<STRING>                  ::=    /’[^\’]’/
                                 | /”[^\”]”/

<IDENTIFIER>              ::=    /[^(\'|\"|fi|if|esle|else|loop|pool|\,
                                 |cluster\[|\])][a-z_]+[a-zA-Z0-9_]*/

<LOOP_STMT>               ::=    “pool” <STMT_LIST> “loop (“ <ASSIGN_STMT>
                                 “;” <OR_TEST> “;” <EXPRESSION> “)”
                                 | “pool” <STMT_LIST> “loop (“ <EXPRESSION> “)”


<FUNCTION_DECL>           ::=    “boj” <STMT_LIST> “job” <IDENTIFIER> “(“
                                 <PARAM_LIST> “)”
                                 | “boj” <STMT_LIST> “job” <IDENTIFIER> “()”

<OP_RELATIONAL>           ::=    “<”
                                 | “<=”
                                 | “>”
                                 | “>=”
                                 | “==”
                                 | “!=”
```

```
<ARG_LIST>              ::=   <ARG_LIST> “,” <ARG>
                              | <ARG>

<ARG>                   ::=   <EXPRESSION>

<PARAM_LIST>            ::=   <PARAM_LIST> “,” <PARAM>
                              | <PARAM>

<PARAM>                 ::=   <EXPRESSION>
```

6. Programkod

flipflop.rb

```ruby

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

require './rdparse.rb'
require './node.rb'

class FlipFlop
 def initialize
  @parser = Parser.new('Flip/Flop') do

   ## LEXER
   # One-row comments
   token(/^#(.)*$/)
   # Multi-rows comments
   token(/^(##[\w\W\s]*##)/)
   # White spaces
   token(/^(\s)/)
   # Specific syntax for the language
   token(/^(scream|yes|no|fi|esle|else|esle fi|cluster|boj|job)/) { |m| m }
   # Operators etc.
   token(/^(\++|\--|\+|\-|\*|\/|\%|\!=|\==|\<=|\>=|\=|\!|\&&|\<|\>|\(|\)|\]|
         \[|\|\||\;|\,)/) { |m| m }
   # Negative floats
   token(/^(-(\d+[.]\d+))/) {|m| m.to_f }
   # Positive floats
   token(/^(\d+[.]\d+)/) {|m| m.to_f }
   # Digits
   token(/^(\d+)/) { |m| m.to_i }
   # Strings that starts with '
   token(/^('[^\']*')/) { |m| m }
   # Strings that starts with "
   token(/^("[^\"]*")/) { |m| m }
   # Variables
   token(/[a-zA-Z0-9_]+/) { |m| m }
   ## !LEXER
```

```
39      start :program do
40       match(:statement_list) {
41        |stmt_list|
42        Program_Node.new(stmt_list)
43       }
44      end
45
46      ## STATEMENTS
47      rule :statement_list do
48       match(:statement_list, :statement) {
49        |stmt_list, stmt|
50        stmt_list << stmt
51       }
52       match(:statement) { |stmt| [stmt] }
53      end
54
55      rule :statement do
56       match(:function_call)
57       match(:print_statement)
58       match(:assign_statement)
59       match(:if_statement)
60       match(:expression)
61       match(:loop_statement)
62       match(:function_declare)
63      end
64
65      ## FUNCTION CALL
66      rule :function_call do
67       match(:identifier, '(', ')') {
68        |name, _, _|
69        FunctionCall_Node.new(name, nil)
70       }
71       match(:identifier, '(', :argument_list, ')') {
72        |name, _, arg_list, _|
73        FunctionCall_Node.new(name, arg_list)
74       }
75      end
76      ## !FUNCTION CALL
77
78      # The first print-statement will print an expression
79      # The second print-statement will subscript a variable or an array and print the value
```

```
80      rule :print_statement do
81       match(:expression, 'scream') {
82        |expr, _|
83        Print_Node.new(expr)
84       }
85       match(:identifier, '[', :integer_value, ']', 'scream') {
86        |ident, _, int_value, _, _|
87        PrintSubscript_Node.new(ident, int_value)
88       }
89      end
90
91      # Assign an expression to a variable
92      rule :assign_statement do
93       match(:expression, '=', :identifier) {
94        |expr, _, ident|
95        AssignValue_Node.new(ident, expr)
96       }
97      end
98
99      # The first if-statement is an if with an else
100     # The second statement is an if with a number of if-else-statements
101     # The third one is just an if
102     rule :if_statement do
103      # If-else
104      match('fi', '(', :expression, ')', :statement_list, 'esle', :statement_list, 'else')
105      {
106       |_, _, expressions, _, stmt_list1, _, stmt_list2, _|
107       IfElse_Node.new(stmt_list1, stmt_list2, expressions)
108      }
109
110      # If-elseif
111      match('fi', '(', :expression, ')', :statement_list, 'esle', :if_statement) {
112       |_, _, expressions, _, stmt_list1, _, stmt_list2|
113       IfElse_Node.new(stmt_list1, stmt_list2, expressions)
114      }
115
116      # If
117      match('fi', '(', :expression, ')', :statement_list, 'if') {
118       |_, _, expressions, _, stmt_list, _|
119       If_Node.new(stmt_list, expressions)
120      }
```

```
121     end
122
123     ## EXPRESSIONS
124     rule :expression do
125      match(:add_one)
126      match(:subtract_one)
127      match(:or_test)
128      match(:array)
129      match('(', :comparison, ')') { |_, comp, _| comp }
130      match(:atom)
131     end
132
133     rule :add_one do
134      match(:identifier, "++") {
135       |ident, _|
136       AddOne_Node.new(ident)
137      }
138     end
139
140     rule :subtract_one do
141      match(:identifier, '--') {
142       |ident, _|
143       SubtractOne_Node.new(ident)
144      }
145     end
146
147     rule :or_test do
148      match(:and_test)
149      match(:or_test, "||", :and_test) {
150       |expr1, operator, expr2|
151       Compound_Node.new(operator, expr1, expr2)
152      }
153     end
154
155     rule :and_test do
156      match(:not_test)
157      match(:and_test, "&&", :not_test) {
158       |expr1, operator, expr2|
159       Compound_Node.new(operator, expr1, expr2)
160      }
161     end
```

```
162
163     rule :not_test do
164      match(:comparison)
165      match("!", :not_test) {
166       |_, expr|
167       NotTest_Node.new(expr)
168      }
169     end
170
171     rule :expression_addition do
172      match(:expression_multiplication)
173      match(:expression_addition, '+', :expression_multiplication) {
174       |expr1, operator, expr2|
175       Compound_Node.new(operator, expr1, expr2)
176      }
177      match(:expression_addition, '-', :expression_multiplication) {
178       |expr1, operator, expr2|
179       Compound_Node.new(operator, expr1, expr2)
180      }
181     end
182
183     rule :expression_multiplication do
184      match(:expression_unary)
185      match(:expression_multiplication, '*', :expression_unary) {
186       |expr1, operator, expr2|
187       Compound_Node.new(operator, expr1, expr2)
188      }
189      match(:expression_multiplication, '/', :expression_unary) {
190       |expr1, operator, expr2|
191       Compound_Node.new(operator, expr1, expr2)
192      }
193      match(:expression_multiplication, '%', :expression_unary) {
194       |expr1, operator, expr2|
195       Compound_Node.new(operator, expr1, expr2)
196      }
197     end
198
199     rule :expression_unary do
200      match(:atom)
201      match('-', :expression_unary) {
202       |_, unary|
```

```
203       unary * -1
204      }
205     end
206
207     ## ARRAY
208     rule :array do
209      match('cluster', '(', :array_values, ')', '=', :identifier) {
210       |_, _, stmt_list, _, _, identifier|
211       ArrayNew_Node.new(identifier, stmt_list)
212      }
213     end
214
215     rule :array_values do
216      match(:atom) { |atom| [atom] }
217      match(:array_values, ',', :atom) {
218       |array_values, _, atom|
219       array_values << atom
220      }
221     end
222     ## !ARRAY
223
224     rule :comparison do
225      match(:expression_addition, :op_relational, :expression_addition) {
226       |expr1, operator, expr2|
227       Compound_Node.new(operator, expr1, expr2)
228      }
229      match(:expression_addition) {
230       |expr|
231       ArithmeticExpr_Node.new(expr)
232      }
233     end
234
235     rule :atom do
236      match(:boolean_value)
237      match(:integer_value)
238      match(:float_value)
239      match(:string_value)
240      match(:identifier)
241     end
242
243     rule :boolean_value do
```

```
244      match("yes") { |bool_value|
245       Boolean_Node.new(bool_value)
246      }
247      match("no") { |bool_value|
248       Boolean_Node.new(bool_value)
249      }
250     end
251
252     rule :integer_value do
253      match(Integer) { |int_value|
254       Integer_Node.new(int_value)
255      }
256     end
257
258     rule :float_value do
259      match(Float) { |float_value|
260       Float_Node.new(float_value)
261      }
262     end
263
264     rule :string_value do
265      match(/^('[^\']*')/) { |string_value|
266       String_Node.new(string_value)
267      }
268      match(/^("[^\"]*")/) { |string_value|
269       String_Node.new(string_value)
270      }
271     end
272
273     # Variable match
274     # The reg-exp is pretty ugly but for some reason we need to have it like this or
275     # the parser won't recognize when we declare our variables
276     rule :identifier do
277      match(/[^(\'|\"|fi|if|esle|else|loop|pool|boj|job|\(|\)|\,|cluster\[|
278           \])][a-z_]+[a-zA-Z0-9_]*/) { |var|
279       Variable_Node.new(var)
280      }
281     end
282     ## !EXPRESSIONS
283
284     ## LOOP
```

```
285
286     # The first statement is for a for-loop.
287     # The second is for a while-loop.
288     rule :loop_statement do
289      # For-loop
290      match("pool", :statement_list, "loop", "(", :assign_statement, ';',
291            :or_test, ';', :expression, ")") {
292       |_, statement_list, _, _, assign_statement, _, or_test, _, expression, _|
293       LoopFor_Node.new(statement_list, assign_statement, or_test, expression)
294      }
295
296      # While-loop
297      match("pool", :statement_list, "loop", "(", :expression, ")") {
298       |_, stmt_list, _, _, expressions, _|
299       LoopWhile_Node.new(stmt_list, expressions)
300      }
301     end
302     ## !LOOP
303
304     ## FUNCTION DECLARATION
305     rule :function_declare do
306      match('boj', :statement_list, 'job', :identifier, '(', :parameter_list, ')') {
307       |_, stmt_list, _, name, _, param_list, _|
308       FunctionDeclare_Node.new(stmt_list, name, param_list)
309      }
310      match('boj', :statement_list, 'job', :identifier, '(', ')') {
311       |_, stmt_list, _, identifier, _, _|
312       FunctionDeclare_Node.new(stmt_list, identifier, nil)
313      }
314     end
315     ## !FUNCTION DECLARATION
316     ## !STATEMENTS
317
318     ## OPERATOR RELATIONAL
319     rule :op_relational do
320      match('<')
321      match('<=')
322      match('>')
323      match('>=')
324      match('==')
325      match('!=')
```

```
326     end
327     ## !OPERATOR RELATIONAL
328
329     rule :argument_list do
330      match(:argument_list, ',', :argument) {
331       |arg_list, _, arg|
332       arg_list + [arg]
333      }
334      match(:argument) { |arg| [arg] }
335     end
336
337     rule :argument do
338      match(:expression)
339     end
340
341     rule :parameter_list do
342      match(:parameter_list, ',', :parameter) {
343       |param_list, _, param|
344       param_list+ [param]
345      }
346      match(:parameter) { |param| [param] }
347     end
348
349     rule :parameter do
350      match(:identifier)
351     end
352    end
353
354    def done(str)
355     ['exit', 'quit', 'q'].include?(str.chomp)
356    end
357
358    def start_man
359     ffMessenger("Current scope: #{@@scope}") if @@ffHelper
360     ffMessenger("Current variable stack: #{@@variables}") if @@ffHelper
361     ffMessenger("Current function stack: #{@@functions}") if @@ffHelper
362
363     print "f/f > "
364     user_input = gets
365
366     if done(user_input) then
```

```
367       ffMessenger("Bye bye!\n\n")
368      else
369       result = @parser.parse(user_input)
370
371       result.evaluate()
372       start_man
373      end
374     end
375
376     def parse_file(filename)
377      @parser.parse(IO.read(filename)).evaluate()
378     end
379
380     def log(state = true)
381      if state
382       @parser.logger.level = Logger::DEBUG
383      else
384       @parser.logger.level = Logger::WARN
385      end
386     end
387    end
388
389    ff = FlipFlop.new
390    ff.log(false)
391
392    if (ARGV.length > 0) then
393     filename = ARGV[0]
394     ff.parse_file(filename)
395    else
396     puts ""
397     84.times { |x| print "*" }; puts "*"
398     print "*", " ".center(83); puts "*"
399     print "*", "FLIP/FLOP 1.0".center(83); puts "*"
400     print "*", " ".center(83); puts "*"
401     print "*", "Dishing out f/f awesomeness since 2012. All rights reserved.
402           Not really.".center(83); puts "*"
403     print "*", "User's manual is located in /Docs.".center(83); puts "*"
      print "*", "Type 'exit', 'quit' or 'q' to exit.".center(83); puts "*"
      print "*", " ".center(83); puts "*"
      84.times { |x| print "*" }; puts "*"
```

```
        ff.start_man
      end
    end
```

node.rb

```ruby

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

@@variables =[{}] # All declared variables. Each index is a separate scope
@@functions = {} # Function names, their nodes and their parameters
@@scope = 0 # Current scope level (index used in @@variables)

@@ffHelper = false #Set to true to see parse flow

# SCOPE HANDLING ------------------------------>

def open_scope
  @@scope += 1
  @@variables.push({})
  ffMessenger("Opened a scope, now at level #{@@scope}") if @@ffHelper
end

def close_scope
  @@variables.pop
  @@scope -= 1
  ffMessenger("Closed a scope, now at level #{@@scope}") if @@ffHelper
end

def add_to_scope(name, value)
  current_scope = @@variables[@@scope]
  current_scope[name] = value
end

def lookup(identifier, hash)
  ffMessenger("Called lookup function") if @@ffHelper
  if hash == @@functions
    hash[identifier]

  elsif hash == @@variables
    i = @@scope
    while(i >= 0)
      ffMessenger("Searching for \"#{identifier}\" at scope #{i}") if @@ffHelper

```

```
42        if @@variables[i].include? (identifier) then
43          return @@variables[i][identifier]
44        else
45          i -= 1
46        end
47      end
48      if @@variables[0][identifier] == nil
49        ffMessenger("There is no variable called #{identifier}.")
50      end
51    end
52  end
53
54  # END SCOPE HANDLING ---------------------------->
55
56  def ffMessenger(str)
57    print "flip/flop says: "; puts str
58  end
59
60  class Program_Node
61    def initialize(_stmt_list)
62      @stmt_list = _stmt_list
63    end
64
65    def evaluate()
66      ffMessenger("Entered Program_Node") if @@ffHelper
67
68      @stmt_list.each do
69        |prog|
70        prog.evaluate()
71      end
72    end
73  end
74
75  class Print_Node
76    def initialize(_expr)
77      @expr = _expr
78    end
79
80    def evaluate()
81      ffMessenger("Entered Print_Node") if @@ffHelper
82
```

```ruby
    puts @expr.evaluate()
  end
end

# Used to print a specific value of an array or a variable
class PrintSubscript_Node
  attr_accessor :name, :subscript

  def initialize(_name, _int)
    @name = _name
    @subscript = _int
  end

  def evaluate()
    ffMessenger("Entered PrintSubscript_Node") if @@ffHelper

    value = lookup(@name.value, @@variables)

    if @subscript.evaluate() == 0 then
      ffMessenger("#{@subscript.evaluate()} is not a valid subscript value.")

    elsif value.class == String || value.class == Array
      if @subscript.evaluate() <= value.size
        puts value[@subscript.evaluate() - 1]
      else
        ffMessenger("#{@subscript.evaluate()} is not a valid subscript value.")
      end

    elsif value.class == Fixnum
      ffMessenger("It is not possible to subscript an Integer.")

    elsif value.class == Float
      ffMessenger("It is not possible to subscript a Float.")

    else
      puts "Oh, what do we have here? A #{value.class}. Guess we forgot to implement
subscripting for that."

    end
  end
end
```

```ruby

# Assign a value to a variable
class AssignValue_Node
  attr_accessor :var_name, :var_expr

  def initialize(_var_name, _var_expr)
    @var_name = _var_name
    @var_expr = _var_expr
  end

  def evaluate()
    ffMessenger("Entered AssignValue_Node") if @@ffHelper

    @@variables[@@scope][@var_name.value] = @var_expr.evaluate()

    ffMessenger("This is the current variable stack: #{@@variables}") if @@ffHelper
  end
end

# Used for if-else and if-elseif statements
class IfElse_Node
  attr_accessor :if_body, :else_body, :expressions

  def initialize(_if_body, _else_body, _expressions)
    @if_body = _if_body
    @else_body = _else_body
    @expressions = _expressions
  end

  def evaluate()
    ffMessenger("Entered IfElse_Node") if @@ffHelper

    if @expressions.evaluate() then
      @if_body.each do
        |if_stmt|
        if_stmt.evaluate()
      end
    else
      if @else_body.class == Array then
        @else_body.each do
          |else_stmt|
```

```
165             else_stmt.evaluate()
166           end
167         else
168           @else_body.evaluate()
169         end
170       end
171     end
172   end
173
174   # Used only for the if-statement
175   class If_Node
176     attr_accessor :if_body, :expressions
177
178     def initialize(_if_body, _expressions)
179       @if_body = _if_body
180       @expressions = _expressions
181     end
182
183     def evaluate()
184       ffMessenger("Entered If_Node") if @@ffHelper
185
186       if @expressions.evaluate() then
187         @if_body.each do
188           |if_stmt|
189           if_stmt.evaluate()
190         end
191       end
192     end
193   end
194
195   class AddOne_Node
196     attr_accessor :var_name
197
198     def initialize(_var_name)
199       @var_name = _var_name
200     end
201
202     def evaluate()
203       ffMessenger("Entered AddOne_Node") if @@ffHelper
204
205       old_value = lookup(@var_name.value, @@variables).to_i
```

```
206       new_value = old_value + 1
207
208       i = @@scope
209       while(i >= 0)
210         ffMessenger("Looking to add 1 to \"#{var_name.value}\" at scope #{i}") if
211   @@ffHelper
212
213         if @@variables[i].include? (@var_name.value) then
214           @@variables[i][@var_name.value] = new_value
215           break
216         else
217           i -= 1
218         end
219       end
220
221       ffMessenger("Added 1 to #{var_name.value} at scope #{i}") if @@ffHelper
222
223     end
224   end
225
226   class SubtractOne_Node
227     attr_accessor :var_name
228
229     def initialize(_var_name)
230       @var_name = _var_name
231     end
232
233     def evaluate()
234       ffMessenger("Entered SubtractOne_Node") if @@ffHelper
235
236       old_value = lookup(@var_name.value, @@variables).to_i
237       new_value = old_value - 1
238
239       i = @@scope
240       while(i >= 0)
241         ffMessenger("Looking to subtract 1 from \"#{var_name.value}\" at scope #{i}") if
242   @@ffHelper
243
244         if @@variables[i].include? (@var_name.value) then
245           @@variables[i][@var_name.value] = new_value
246           break
```

```
247         else
248           i -= 1
249     end
250       end
251
252       ffMessenger("Subtracted 1 from #{var_name.value}") if @@ffHelper
253
254     end
255   end
256
257   # Compound_Node is used for several different operations
258   # It is used for predicate expressions and arithmetic expression
259   class Compound_Node
260     attr_accessor :operator, :value1, :value2
261
262     def initialize(_operator, _value1, _value2)
263       @operator = _operator
264       @value1 = _value1
265       @value2 = _value2
266     end
267
268     def evaluate()
269       ffMessenger("Entered Compound_Node") if @@ffHelper
270
271       if value1.evaluate().class == String and value2.evaluate().class == String
272         instance_eval("'#{value1.evaluate()}' #{operator} '#{value2.evaluate()}'")
273       end
274
275       instance_eval("#{value1.evaluate()} #{operator} #{value2.evaluate()}")
276     end
277   end
278
279   # Returns true or false depending on the value
280   class NotTest_Node
281     def initialize(_value)
282       @value = _value
283     end
284
285     def evaluate()
286       ffMessenger("Entered NotTest_Node") if @@ffHelper
287
```

```
288       return (not @value.evaluate())
289     end
290   end
291
292   class ArrayNew_Node
293     attr_accessor :array_name, :values
294
295     def initialize(_array_name, _values)
296       @array_name = _array_name
297       @values = _values
298     end
299
300     def evaluate()
301       ffMessenger("Entered ArrayNew_Node") if @@ffHelper
302
303       r_array = []
304       @values.each do
305         |array_values|
306         r_array << array_values.evaluate()
307         @@variables[@@scope][@array_name.value] = r_array
308
309         puts @@variables if @@ffHelper
310       end
311     end
312   end
313
314   =begin NOT IMPLEMENTED YET
315   class ArrayIndex_Node
316     def initialize(_array_name, _get_index)
317       @array_name = _array_name
318       @get_index = _get_index
319     end
320
321     def evaluate()
322       ffMessenger("Entered ArrayIndex_Node") if @@ffHelper
323
324       puts @@arrayHash[@array_name]
325     end
326   end
327
328   class ArraySize_Node
```

```
329    def initialize(_array_name)
330      @array_name = _array_name
331    end
332
333    def evaluate()
334      puts @array_name.value
335    end
336  end
337  =end
338
339  class ArithmeticExpr_Node
340    def initialize(_expr)
341      @expr = _expr
342    end
343
344    def evaluate()
345      ffMessenger("Entered ArithmeticExpr_Node") if @@ffHelper
346
347      return @expr.evaluate()
348    end
349
350  end
351
352  # NOT PROPERLY IMPLEMENTED YET. SHOULD RETURN 'yes' AND 'no'.
353  class Boolean_Node
354    def initialize(_expr)
355      @expr = _expr
356    end
357
358    def evaluate()
359      ffMessenger("Entered Boolean_Node") if @@ffHelper
360
361      case @expr
362      when 'yes'
363        return true
364      when 'no'
365        return false
366      end
367    end
368  end
369
```

```ruby
class Integer_Node
  def initialize(_value)
    @value = _value
  end

  def evaluate()
    ffMessenger("Entered Integer_Node") if @@ffHelper

    return @value
  end
end

class Float_Node
  def initialize(_value)
    @value = _value
  end

  def evaluate()
    ffMessenger("Entered Float_Node") if @@ffHelper
    return @value
  end
end

class String_Node
  def initialize(_value)
    @value = _value
  end

  def evaluate()
    ffMessenger("Entered String_Node") if @@ffHelper

    return @value.delete "\"\'" # Deletes quotation marks to prevent subscript issues
  end
end

class Variable_Node
  attr_accessor :var

  def initialize(_var)
    @var = _var
  end
```

```
411
412   def evaluate()
413     ffMessenger("Entered Variable_Node") if @@ffHelper
414
415     return lookup(@var, @@variables)
416
417   end
418 end
419
420 class LoopFor_Node
421   attr_accessor :stmt_list, :assign_stmt, :or_test, :expr
422
423   def initialize(_stmt_list, _assign_stmt, _or_test, _expr)
424     @stmt_list = _stmt_list
425     @assign_stmt = _assign_stmt
426     @or_test = _or_test
427     @expr = _expr
428   end
429
430   def evaluate()
431     ffMessenger("Entered LoopFor_Node") if @@ffHelper
432
433     open_scope()
434     @assign_stmt.evaluate()
435     ffMessenger("Created and assigned loop variable") if @@ffHelper
436
437     while @or_test.evaluate() do
438   ffMessenger("Executing loop statements...") if @@ffHelper
439   @stmt_list.each do
440       |stmt|
441       stmt.evaluate()
442   end
443   @expr.evaluate()
444     end
445     close_scope()
446   end
447 end
448
449 class LoopWhile_Node
450   attr_accessor :stmt_list, :expressions
451
```

```
452   def initialize(_stmt_list, _expressions)
453     @stmt_list = _stmt_list
454     @expressions = _expressions
455   end
456
457   def evaluate()
458     ffMessenger("Entered LoopWhile_Node") if @@ffHelper
459
460     open_scope()
461     while @expressions.evaluate() do
462       @stmt_list.each do
463         |stmt|
464         stmt.evaluate()
465       end
466     end
467     close_scope()
468   end
469 end
470
471 class FunctionDeclare_Node
472     attr_accessor :stmt_list, :identifier, :param_list
473
474   def initialize(_stmt_list, _identifier, _param_list)
475     @stmt_list = _stmt_list
476     @identifier = _identifier
477     @param_list = _param_list
478     end
479
480   def evaluate()
481     ffMessenger("Entered FunctionDeclare_Node") if @@ffHelper
482
483     ffMessenger("param_list: #{@param_list.inspect}") if @@ffHelper
484
485     @@functions[@identifier.value] = [@stmt_list, @param_list]
486   end
487 end
488
489 class FunctionCall_Node
490   attr_accessor :name, :arg_list
491
492   def initialize(_name, _arg_list)
```

```
493       @name = _name
494       @arg_list = _arg_list
495     end
496
497     def evaluate()
498       ffMessenger("Entered FunctionCall_Node") if @@ffHelper
499
500       open_scope()
501
502       ffMessenger("These are the #{@name.value} parameters:") if @@ffHelper
503
504       if @@functions[@name.value][1] != nil then
505         ffMessenger("#{@@functions[@name.value][1].inspect}") if @@ffHelper
506       else
507         ffMessenger("#{@name.value} has no parameters.") if @@ffHelper
508       end
509
510       params = @@functions[@name.value][1]
511       if params.size != @arg_list.size then
512         ffMessenger("Passed #{@arg_list.size} arguments to #{@name.value}, but
513 #{params.size} are required.")
514       end
515
516       # Add arguments to scope
517       params.each_with_index do |param, i|
518         name = param.value
519         arg = @arg_list[i]
520         add_to_scope(name, arg.evaluate)
521       end
522
523       # Executes the function body
524       stmt_list = @@functions[@name.value][0]
525       stmt_list.each do |stmt|
          stmt.evaluate()
        end
        ffMessenger("The function body executed ok.") if @@ffHelper

        close_scope()
      end
    end
```

run.rb

```
1
2
3
4
load './flipflop.rb'
```