{TwenTy Twelve}

TTT

Linköpings universitet
Innovativ Programmering
TDP019 - Projekt: Datorspråk

Patrik Ottosson, patot329@student.liu.se
Simon Erlandsson, simer656@student.liu.se

Utskriftdatum: 2012-05-23
Examinator: Anders Haraldsson, IDA

Projektdokumentation - TDP019

Sammanfattning

Programmeringsspråket TwenTy Twelve eller TTT är ett resultat från kursen TDP019 på programmet Innovativ Programmering på Linköpings universitet. Där fick vi i uppgift att implementera ett eget programmeringsspråk. Vi skapade TTT med hjälp av Ruby och vi inspirerades av Ruby och C++. TTT är ett imperativt språk. Det speciella med vårt språk är att det är typat. Vi har typning på allt utom hashar, vilket kan ses som att det är svårt språk för nybörjare, men vi har implementerat en speciell typ i språket som heter 'ALL' . Denna typ kan lagra alla typer av information. Det ger nybörjare chansen att lära sig språket utan att behöva bry sig om typningen. Typningen har gett en rolig egenskap till funktioner också. Det har varit en bra sak att kunna ha en ALL som inparameter. Det är bra om man t.ex. ska göra en utskriftsfunktion så vill man inte ha någon typ utan bara kunna skriva ut det som kommer in till funktionen. TTT är implementerat med hjälp av ett vertyg som heter RDparser som är skrivit i Ruby.

Innehållsförteckning

1. Inledning 5

1.1 Syfte 5

1.2 Introduktion 5

1.3 Målgrupp 5

2. Användarhandledning 5

2.1 Installation 5

2.2 Konstruktioner 5

2.2.1 Datatyper 5

2.2.2 Språkstruktur 6

2.2.3 Tilldelning och variabler 6

2.2.4 Operatorer 6

2.2.5 In- och utmatning 8

2.2.6 Villkorsatser 8

2.2.7 Repetitionssatser 9

2.2.7.1 For-loopar 9

2.2.7.2 Each-loopar 9

2.2.7.3 While-loopar 10

2.2.8 Listor 11

2.2.9 Hash tabeller 11

2.2.10 Done 12

2.2.11 Funktioner 12

2.2.12 Hantering av räckvidd 12

2.3 Inkludera i intepretatorn 13

2.4 Kommentarer 13

3. Systemdokumentation 14

3.1 Översikt 14

3.1.1 Lexikalisk analys 14

3.1.2 Parsing 14

3.1.3 Abstrakt syntaxträd 14

3.2 Kodstandard 15

3.4 Grammatik 15

4. Reflektion 17

5. Bilagor 18

TTT.rb 18

TTTBasic.rb 20

TTTClasses.rb 24

parser.rb 35

rdparser.rb 41

1. Inledning

Det här projektet genomfördes på IP-programmet år 1, i kursen TDP019 Projekt: Datorspråk, under andra terminen. Rapporten består av fyra delar; inledning, användarhandledning, system-dokumentation samt reflektion.

1.1 Syfte

Syftet med projektet var att vi skulle lära oss hur ett programspråk är uppbyggt. Genom att skapa vårt eget programspråk lärde vi oss hur programkod körs och hur den tolkas.

1.2 Introduktion

Vår idé var att kombinera delar som vi uppskattar från C++ och Ruby: Rubys lätta syntax och frihet samt typingen från C++. Vi har även valt att avsluta de flesta satser i vårt språk genom att skriva / följt av vilken konstruktion som används. Detta kan jämföras med användandet av end i Ruby samt {} i C++.

1.3 Målgrupp

Vårt språk är ett enkelt språk som riktar sig mot nybörjare som vill börja med ett språk där typning är möjligt, men samtidigt är fritt genom att det finns en datatyp som fungerar som alla datatyper. Detta leder till att man själv får välja hur strikt man vill vara med typning. Det ska även vara lätt att lära sig och förstå hur det används genom enkel och förståelig syntax. Genom att kombinera dessa två saker ska det ge nybörjare en bra start på löpbandet mot mer avancerade program och programspråk.

2. Användarhandledning

2.1 Installation

Installera Ruby; Starta därefter en terminal och navigera till mappen där TTT.rb är sparad; Skriv sedan ”ruby TTT.rb” för att starta interpretatorn.

2.2 Konstruktioner

De konstruktioner som finns i vårt språk är villkorssatser, repetitionssatser, datatyper, operatorer och funktioner.

2.2.1 Datatyper

I vårt språk finns datatyperna NUM, STR och BOOL, som motsvarar float, string och boolean i många andra språk. Vi har även listor och hashar för att lagra information med. Datatyperna är typade i vårt språk förutom hashar. Typningen gör att det blir lättare för användaren att inte sätta fel typer och få error vid uträkningar. Men vi har en typ som heter ALL. Denna typ innehåller NUM, STR och BOOL i ett. ALL ger användaren möjligheten att bortse från typning. Det medför både för- och nackdelar. Vi har även listor och hashar i vårt språk som vi skriver mera om under 2.2.8 samt 2.2.9.

2.2.2 Språkstruktur

I vårt språk har vi valt att ha nyckelordstruktur med ett start- och ett slutnyckelord. Detta gäller alla våra block och deklarationer.

2.2.3 Tilldelning och variabler

En variabel ska bestå av tecknen A-Z,a-z,0-9 samt "_" och en variabel måste börja på en stor bokstav och vara minst ett tecken långt. Alla variablar ska vid deklaration deklareras med en datatyp. Detta är kriterierna för variablarna i språket. En deklaration inleder man med att skriva DECL och avslutar med /DECL. När en variabel är deklarerad kan man tilldela ett nytt värde till den genom att skriva variabelns namn följt av : (kolon) och därefter värdet man vill tilldela variabeln.

Tilldelning och anrop av variabel i vårt språk kan se ut så här:

```
>> DECL <STR> Hej: ”hejsan” /DECL
”hejsan”
>> Hej
hejsan
>> Hej: ”Tja”
”Tja”
>> DECL <NUM> Tal: 3 /DECL
3
>> Tal
3
```

2.2.4 Operatorer

För aritmetiska och andra olika operationer i språket behöver vi tre olika kategorier i vårt språk. Dessa tre kategorier är aritmetiska operatorer, logiska operatorer och jämförelseoperatorer. Vi har använt oss av de vanliga matematiska prioriteringarna i vårt språk.

- Aritmetiska operatorer: + , - , * , / , och %
- Logiska operatorer: &&, || , !, AND, OR och NOT
- Jämförelseoperatorer: ==, =/=, > , >= , < och <=

Aritmetiska operationer i vårt språk kan se ut så här:

```
>> 3+4
== 7
>> (3-1)*(1+1)
== 4
>> 8 / 4
== 2
```

Logiska operationer i vårt språk kan se ut så här:

```
>> TRUE AND FALSE
== FALSE
>> TRUE OR FALSE
== TRUE
>> NOT FALSE
== TRUE
>> NOT TRUE
== FALSE
>> 1 AND 2
== 2
>> 1 OR 2
== 1
```

Jämförelseoperationer i vårt språk kan se ut så här:

```
>> TRUE == FALSE
== FALSE
>> 4 > 5
== FALSE
>> 4 >= 4
== TRUE
>> 4 < 5
== TRUE
>> 4 <= 3
== FALSE
```

2.2.5 In- och utmatning

För in- och utmatning används READ respektive PRINT. READ väntar på inmatning från användaren och sparar värdet i den givna variabeln. PRINT tar ett uttryck och skriver ut resultatet.

In och utmatning i vårt språk kan se ut så här:

```
>> PRINT ”HELLO WORLD” /PRINT
== HELLO WORLD
>> DECL <STR> Indata /DECL
>> READ Indata /READ
<< JAG MATAR IN DETTA
>> PRINT Indata /PRINT
== JAG MATAR IN DETTA
```

2.2.6 Villkorsatser

If-satser är ett viktigt styrvertyg i språket. If-satser i språket är som helt vanliga if-satser som finns i andra språk. If-satsen består av tre delar där de två sista är valbara. Dessa tre är IF, ELSEIF och ELSE. IF och ELSEIF måste ha ett villkor efter sig. Ett villkor som ska bli sant eller falskt. ELSE är till för om IF eller ELSEIF inte är sanna.

If-satser i vårt språk kan se ut så här:

```
>> IF(3<4) PRINT ”YES” /PRINT /IF
== ”YES”

>> IF( NOT(3==4)) PRINT ”NOT” /PRINT /IF
== ”NOT”

>> IF(5<3) PRINT ”3 IS BIG” /PRINT ELSE PRINT ”5 IS BIG” /IF
== ”5 IS BIG”
>> DECL <NUM> Tal: 4 /DECL
== 4
>> IF (Tal < 3)
..      PRINT ”SMALLER THEN 3” /PRINT
..  ELSEIF ( Tal > 3 )
..      PRINT ”BIGGER THEN 3” /PRINT
..  ELSE
..      PRINT ”THE SAME” /PRINT
..  /IF
== ”BIGGER THEN 3”
```

2.2.7 Repetitionssatser

Vi har tre olika repetitionssatser i vårt språk: for-loop, each-loop och while-loop. While-loopen är en vanlig och traditionell loop som ser ut som de vanliga While-loopar gör i andra språk. For- och each-loopar i vårt språk ser dock inte ut som andra språk. Vår each-loop är en iterator för listor. Den itererar elementen i en lista och så kan man göra något med det elementet. For-loopar i vårt språk itererar från ett startnummer till ett slutnummer plus att man får välja ökningsgrad, nästan som C++ for-loop, dock mer begränsad.

2.2.7.1 For-loopar

For-loopar i språket har fyra värden: en styrvariabel, två värden som skapar ett intervall och ett värde som bestämmer ökning på styrvariabeln för varje varv.

For-loopar i vårt språk kan se ut så här:

```
>> FOR (<NUM> I  IN 1 TO 3 INCBY 1) PRINT I /PRINT /FOR
== 1
== 2
== 3

>> FOR (<NUM> I IN 1 TO 5 INCBY 2) PRINT I /PRINT /FOR
== 1
== 3
== 5
```

2.2.7.2 Each-loopar

Each-loopar är en loop som tar en lista som parameter och tilldelar en variabel ett värde i taget tills alla element i listan är slut. En each-loop i språket har två värden: en variabel som tilldelas ett element i taget som en loop. Sedan har vi själva listan som innehåller elementen som man vill åt.

Each-loopar i vårt språk kan se ut så här:

```
>> LIST <NUM> Lista :  [ 1,2,3 ] /LIST
== [1,2,3]
>> EACH ( <NUM> I IN Lista) PRINT I /PRINT /EACH
== 1
== 2
== 3
>> EACH ( <NUM> I IN Lista) PRINT I+1 /PRINT /EACH
== 2
== 3
== 4
```

2.2.7.3 While-loopar

While-loopar i språket är som de flesta språks while-loopar. Den har ett villkorsuttryck som i varje varv beräknas och om värdet är sant körs det innanför while-loopen, annars går programmet vidare.

While-loopar i vårt språk kan se ut så här:

```
>> DECL <NUM> Tal: 1 /DECL
== 1
>> WHILE (Tal < 5) PRINT Tal /PRINT  Tal: Tal +1  /WHILE
== 1
== 2
== 3
== 4
>> DECL <BOOL> Run: TRUE /DECL
== TRUE
>> WHILE (Run)
..   PRINT ”ONE TIME” /PRINT DECL  RUN:FALSE
..   /WHILE
== ONE TIME
```

2.2.8 Listor

Listor i språket finns för att lagra större mängder information. Informationen lagras i element som ligger i följd efter varandra. Listan är typad.

Listor i vårt språk kan se ut så här:

```
>> LIST <NUM> Lista :  [ 1,2,3 ] /LIST
== [1,2,3]
>> Lista[0]
== 1
>> Lista[1]
== 2
>> Lista[]
== [1,2,3]
>> LIST <STR>  Ord : [”one”,”two”,”end”] /LIST
== [”one”,”two”,”end”]
>> Ord[1]
== two
>> Ord[-1]
== end
>> REMOVE Ord[1]
== ”two”
>> ADD Ord : ”three”
== [”one”,”end”,”three”]
>> EMPTY Ord
== FALSE
```

2.2.9 Hash tabeller

Hash tabeller i språket finns för att lagra information med en nyckel och ett värde.

Hash tabeller i vårt språk kan se ut så här:

```
>> HASH Tal : {”one”>>1,”two”>>2} /HASH
== two2one1
>> Tal {”one”}
== 1
>> Tal {”two”}
== 2
>> ADD Tal {”three”>>3}
== three3
>> REMOVE Tal {”two”}
== 2
```

2.2.10 Done

Done i vårt språk är en retur-sats. Den returnerar det givna blocket mellan DONE och /DONE. Done används huvudsakligen till funktioner vilket vi tar upp i nästa stycke. Done kan användas till att avbryta loopar med.

Done i vårt språk kan se ut så här:

```
>> DONE ”Hej” /DONE
”Hej”
>> DONE 2+4 /DONE
6.0
```

2.2.11 Funktioner

Funktioner i språket skrivs med tre delar: namn på funktionen, parameterlista samt blocket med vad som ska utföras. Namnet på en funktion ska bara bestå av stora bokstäver samt _ och namnet måste vara minst ett tecken långt. Parameterlistan kan bestå av ett antal typade parametrar. En funktion börjar med FUNCTION och avslutas med /FUNCTION

Funktioner i vårt språk kan se ut så här:

```
>> FUNCTION SAY (<STR> Strn) PRINT Strn  /PRINT /FUNCTION
>> SAY(”GOOD FUNCTION!”)
== GOOD FUNCTION!
>>
>> FUNCTION ADD(<NUM> Number1,<NUM> Number2)
..       DONE Number1 + Number2 /DONE
.. /FUNKTION
>> PRINT ADD(1,2) /PRINT
== 3
>>
```

2.2.12 Hantering av räckvidd

Räckvidd, eller scope är till för att hantera vilken räckvidd en variabel har till sin omgivning. En variabel har olika räckvidd beroende på var den är deklarerad. En variabel som tilldelas i en While-loop existerar bara inom den while-loopen. Dess räckvidd sträcker sig inte utanför den satsen. Hanteringen av räckvidd är implementerat med två globala listor och en global variabel. Den första listan innehåller scopes och är till för att hantera sparning av variabler för ett visst scope. Ett scope skapas varje gång man träder in i ett nytt block och försvinner när man träder ur blocket. Varje

scope eller element i listan är en hash som innehåller alla de variabler som existerar inom det scopet. Den andra listan är till för att hantering av variabler inom funktioner ska fungera, den håller reda på vilket nuvarande basescope som gäller. Listan fungerar som en stack (det läggs på och tas bort från toppen). Vi använder basescope för att variabler utanför en funktion inte ska existera inom den funktionen. Den globala variabeln lagrar det nuvarande scopet. När språket letar efter en variabel så börjar den vid det scope den har för tillfället och går ner till sista basescope som finns i scope_base. Hittas ingen variabel finns den inte deklarerad.

2.3 Inkludera i intepretatorn

För att inkludera och köra ett program som har gjorts i en fil använder man sig av intepretatorn och laddar in filen där för att köras där.

Ladda ett program i vårt språk ser t.ex. ut så här:

```
>> INCLUDE Helloworld.ttt
>> HELLOWORLD()
== HELLO WORLD!
```

2.4 Kommentarer

Kommentarer i språket skrivs genom att först skriva # (nummertecken) följt av en kommentar följt av /# (slash nummertecken).

Kommentarer i vårt språk kan se ut så här:

```
>> # En for-lopp som skriver ut alla element i listan. /#
>> LIST Lista<NUM> :  [ 1,2,3 ] /LIST
== [1,2,3]
>> FOR (<NUM> I: /<NUM> IN 0 TO 2 INCBY 1)
..  PRINT Lista[ i ] /PRINT #skriver ut element i lista /#
.. /FOR
>> #Denna kod fungerar ej så kommenterat ut allt.
>> For (<NUM> I: /<NUM> IN 0 TO 2 INCBY 1)
..  PRINT Lista[ i ] /PRINT #skriver ut element i lista /#
.. /for
>> Denna for-loop fungerar ej/#
>>
```

3. Systemdokumentation

3.1 Översikt

Vi använder oss av RDparsern för att göra en lexikalisk analys och parsning av den skrivna TTT-koden. Detta skapar ett abstrakt syntaxträd med noder som är objekt av de klasser som definierats för varje språkkonstruktion. Vid interpreteringen traverseras detta syntaxträd.

3.1.1 Lexikalisk analys

RDparsern börjar med den lexikaliska analysen. Där skapas tokens som är en sekvens av tecken, ofta beskrivna i form av ett reguljärt uttryck. När RDparsern börjar läsa igenom koden så gör den om koden först till nyckelorden som vi har sedan strängar och därefter olika tal och sist alla övriga tecken. När den lexikaliska analysen är gjord skickas en lista med alla token till parsern.

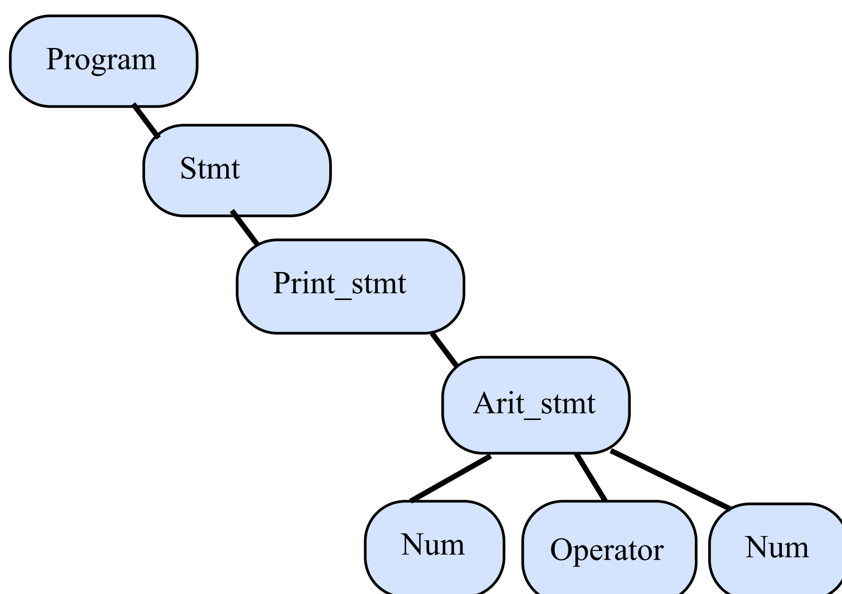
3.1.2 Parsing

RDParsern börjar sin parsning av den sekvens token, som den lexikaliska delen gav genom att matcha token med de reglerna som finns i grammatiken. Med dessa regler parsar man och bygger ett abstrakt syntaxträd.

3.1.3 Abstrakt syntaxträd

När RDparsern har parsat och matchat en konstruktion som t.ex en sträng så skapas ett objekt av klassen STR_C. Alla konstruktioner i språket har en klass som den kan skapa ett objekt av. Alla dessa klasser har en funktion som heter Eval. Detta gör det lätt att exekvera hela trädstruktionen som har skapats i uppbyggnaden vid parsningen. Vi kör Eval på översta objektet som i sin tur aktiverar Eval på sina objekt under sig. Detta sker tills vi når ända ner till sista grenen i trädet och då returneras det uppnådda resultatet.

Exempel på hur trädet kan se ut i vårt språk:



3.2 Kodstandard

TTT följer ingen speciell kodstandard. Vi har därför valt en del kodkonventioner som t.ex. att ha stora bokstäver i funktioner och att variabler ska börja med stor bokstav samt att block har start och slut som IF och /IF etc.

3.4 Grammatik

Grammatiken består av BNF-regler.

```
<program>::= <stmt_list>

<stmt_list>::=  <stmt><stmt_list>
              |  <stmt>

<stmt>::=  <return_stmt>
          |   <io_stmt>
          |   <sel_stmt>
          |   <iter_stmt>
          |   <data_stmt>
          |   <assign_stmt>
          |   <function_stmt>
          |   <expr>

<function_stmt::= <function_def> | <function_call>

<function_def>::= FUNCTION <func_name> ’(’<parameter_list>*’)’
                  <stmt_list> /FUNCTION

<function_call>::= <func_name> ’(’<argument_list>*’)’

<argument_list>::=  <stmt>
                   | <stmt>, <argument_list>

<parameter_list>::= ’<’ <type> ’>’ <var_dec>
                   | ’<’ <type> ’>’ <var_dec>, <parameter_list>

<comments>::= # comments /#

<sel_stmt>::=   IF ’(’ <expr> ’)’  <stmt_list> ELSE <stmt_list> /IF
      |   IF ’(’ <expr> ’)’  <stmt_list> ELSE <elseif> /IF
      |   IF ’(’ <expr> ’)’  <stmt_list> /IF
<elseif>::=  IF ’(’ <expr> ’)‘ <stmt_list>  ELSE <elseif>
      |   IF ’(’ <expr> ’)‘ <stmt_list>  ELSE <stmt_list>
      |   IF ’(’ <expr> ’)’ <stmt_list>

<iter_stmt>::= WHILE ’(’ <expr> ’)’<stmt_list>/WHILE
      |   FOR ’(’<iter_var> IN <num> TO <num> INCBY <num>  ’)’ <stmt_list> /FOR
      |   EACH ’(’<iter_var> IN <var_dec> ’)’ <stmt_list> /EACH

<iter_var>::= <’ <type> ’>’ <var_dec>

<data_stmt>::= EMPTY <var_dec>
             | REMOVE <var_dec> ’[’ <NUM> ’]’
             | REMOVE <var_dec> ’{’ <STR> ’}’
             | ADD <var_dec> <list>
             | ADD <var_dec> : <hash>

<list>::= ’[’ <type_list> ’]’ | ’[’  ’]’
<hash>::=’{’ <type_hash> ’}’ | {’  ’}’

<type_list>::= <type> | <type_list> ’,’ <type>
<type_hash>::= <str> ’>’ <type> | <type_hash> ’,’<str> ’>’ <type>
```

```
<assign_stmt>::=   DECL '<' <type> '>' <var_dec>: /DECL
                    |  DECL '<' <type> '>' <var_dec>: <expr> /DECL
                    |  HASH <var_dec> : <hash> /HASH
                    |  LIST '<' <type>'>' <var_dec>: <list> /LIST


<io_stmt>::=   <print_stmt>
            | <read_stmt>

<print_stmt>::= PRINT <stmt_list> /PRINT

<read_stmt>::= READ <var_dec> /READ | READ '<' <type> '>' <var_dec> /READ

<return_stmt>::= DONE  <stmt_list> /DONE

<expr>::= <expr> <operator_a> <term>
          | <term>

<term>::= <term> <operator_b> <log>
          | <log>

<log>::= <log> <log_operator> <comp>
         | <log_operator_not> <comp>
         | <comp>

<comp>::= <comp> <comp_operator> <factor>
        | <factor>

<factor>::= <function_call>
             | (<expr>)
             | <type>
             | <var_dec>
             | <data>
             | <list>
             | <hash>

<data>::= <var_dec> '[' <num> ']'    // call list
          | <var_dec> '[' <var_dec> ']'    // call list
          | <var_dec> '{' <str> '}'    // call hash
          | <var_dec> '{' <var_dec> '}'    // call hash

<comp_operator>::= == | =/= | > | >= | < | <=

<log_operator>::= && | '||' | AND | OR
<log_operator_not>::= ! | NOT

<operator_a>::=  + | -
<operator_b>::=  * | / |%

<type>::= <num>
           |  <str>
           |  <bool>
           |  <all>

<num>::=   Float
      |  '-' Float

<str>::= sequence /"[\w\s!\?]*"/

<var_dec>::= sequence /[A-Z][a_z0-9_]*/

<func_name>::= sequence /[A-Z_]+/

<bool>::= TRUE | FALSE
```

4. Reflektion

Vi planerade och lade upp vårt språk så att det skulle vara en lätt syntax, tilltalande och skapat för nybörjare. Vi valde ändå att göra ett språk som är typat men ändrade på datatypen hash mot slutet. Vi hade även vår each som vi tänkte skulle likna en for-loop, men vi ändrade den till att vara en iterator för listor.

Vi hade satt upp som mål att klara typning och att vårt språk skulle klara av rekursion (funktioner som kan anropa sig själva). Språket skulle klara av fibonacci samt fakultet som omfattar rekursion. Vi fick problem i slutet med vår rekursion genom att prioriteten av matchningarna inte var rätt satta vilket vi ändrade. Det vi har haft störst problem med har varit att matchningen och typningen skulle fungera. Parsern hade svårt att matcha våra tokens med våra regler. Vi fick ändra till symboler och liknande för matchningen skulle fungera. Problematiken berodde troligtvis på hur RDparsern är uppbyggd. Komplikationen vi hade med typningen var att vi sparade all information i ett objekt. Detta gav oss massor med error och den klarade inte av rekursion då för allt blev samma objekt. Vi löste detta genom att lägga allt i en lista, först värdet och sedan typen.

Totalt sett är vi nöjda med resultatet. Vi lyckades med typning samt rekursion. Vi har lagt ner mycket tid på att få det resultat vi ville ha. En period kändes det som om vi hade tagit på oss för mycket. Dessa tankar kom när vår typning inte ville fungera. Vi känner att vi har lärt oss mycket mer om hur en interpretator fungerar och hur program tolkas samt vad en token är för något. Att jobba med en trädstruktur har varit intressant och kunskapsgivande.

5. Bilagor

Språket består av dessa filer: TTT.rb, TTTBasic.rb, TTTClasses.rb, parser.rb, rdparser.rb.

TTT.rb

```
#!/usr/bin/env ruby
require './TTTClasses.rb'
require './TTTBasic.rb'
require './parser.rb'
#Made in ruby version 1.8.7
class TTT
	#exit funktion
	def done(str)
	["/QUIT","/EXIT",""].include?(str.chomp)
	end
	def sats(str)
		return_value = false
		if (str != "\n" and str != nil)
			if str =~ /^FOR|^WHILE|^IF|^EACH|^FUNCTION/
				return_value = true
			end
		end
		return return_value
	end
	def endsats(str)
		return_value = false
		if (str != "\n" and str != nil)
			if str =~ /\/FOR|\/WHILE|\/IF|\/EACH|\/FUNCTION/
				return_value = true
			end
		end
		return return_value
	end

	def loadfile(str)
		return_value = false
		if (str != "\n" and str != nil)
			litterals = str.split()
			for litteral in litterals do
				if ["INCLUDE"].include?(litteral.strip.chomp)
					return_value = true
				end
			end
		end
		return return_value
	end
	#huvud loopen
	def program
	print ">> "
	str = gets
	satsstring = ""
		if done(str)
		puts "Terminating progress"
```

```
            elsif loadfile(str)
                  str = str.gsub(/INCLUDE/,"").strip.chomp
                  out = ""
                  if File.exist? str
                        File.open(str, 'r') do |f|
                        str = f.readlines
                        out = str.join
                        end
                        "#{@TTTParser.parse out}"
                  else
                        puts ">> no file found"
                  end
                  program
            elsif sats(str)
                  endcount = 0
                  run = true
                  while (run == true)
                        literals = str.split
                        for literal in 0..literals.length do
                              if sats(literals[literal])
                                    endcount +=1
                              end
                              if endsats(literals[literal])
                                    endcount -=1
                              end
                        end
                        satsstring += str
                        if (endcount == 0)
                              run = false
                              break
                        end
                        if endcount > 0
                              print ".. "
                        else
                              print ">> "
                        end
                        str = gets
                  end
                  puts "#{@TTTParser.parse satsstring}"
                  program
            else
                  puts "#{@TTTParser.parse str}"
            program
            end
      end
end
TTT.new.program
```

TTTBasic.rb

```
#!/usr/bin/env ruby
#Made in ruby version 1.8.7
###################### NUM ##############################
class NUM_C
      attr_accessor :value,:type
      def initialize (value)
            @value = value
            @type = :NUM
      end
      def eval()
            return @value
      end
end
###################### STR ##############################
class STR_C
      attr_accessor :value,:type
      def initialize (value)
            @value = value
            @type = :STR
      end
      def eval()
            return @value
      end
end
###################### BOOL ##############################
class BOOL_C
      attr_accessor :value,:type
      def initialize (value)
      #because we don't use ruby's true we convert it to our symbol instead
            if value == true or value == "TRUE"
                  value = :TRUE
            elsif value == false or value == "FALSE"
                  value = :FALSE
            end
            @value = value
            @type = :BOOL
      end
      def eval()
            return @value
      end
end
###################### LIST ##############################
class LIST_C
      attr_accessor :list,:type
      def initialize(list)
            @list = list
            @type = :Array
      end

      def eval
            return @list
      end
```

```
end
####################### LIST GET ##############################
class LIST_GET
      attr_accessor :list,:index
      def initialize(var, index)
            @index = index
            @list = var

      end
      def eval
            if @index != nil
                  return @list.eval[@index.eval]
            else
                  return @list.eval
            end
      end
end
####################### LIST ADD ##############################
class LIST_ADD_C
      attr_accessor :list,:value
      def initialize(list,value)
            @list = list
            @value = value
      end
      def get_type(i)
            return (@@variables[i][@list.name])[1]
      end
      def eval
            value_type = :NUM if (@value.eval[0].kind_of?Float)
            value_type = :STR if (@value.eval[0].kind_of?String)
            type_error = false
            i = @@scope_base.last
            puts "#{@expr} is stored" if (@@Debug)
      while(i<=@@scope)
                  if @@variables[i][@list.name] != nil
                        @type = get_type(i)
                        if(value_type==@type and @type != :ALL)
                              @value.eval.each do |element|
                                    if element.class != @value.eval[0].class
                                          type_error = true
                                    end
                              end
                        elsif(@type != :ALL)
                              type_error = true
                        end
                  end
                  if (type_error)
                        puts "Wrong type definition"
                  return :FALSE
                  end
                  if @@variables[i][@list.name] != nil
                        @value.eval.each do |element|
                              @@variables[i][@list.name][0].push(element)
                        end
                  end
```

```
				i+=1
		end
		@value.eval
		end
end
####################### LIST REMOVE ##############################
class LIST_REMOVE_C
		attr_accessor :list,:value
		def initialize(list,value)
			@list = list
			@value = value
		end
		def eval
			@list.eval.delete_at(@value.eval)
		end
end
####################### HASH ##############################
class HASH_C
		attr_accessor :hash,:type
		def initialize(hash)
			@hash = hash
			@type = :Hash
		end

		def eval
			return @hash
		end
end
####################### HASH GET ##############################
class HASH_GET
		attr_accessor :hash,:index
		def initialize(var, index)
			@index = index
			@hash = var
		end
		def eval
			if @index != nil
				return @hash.eval[@index.eval]
			else
				return @hash.eval
			end
		end
end
####################### HASH ADD ##############################
class HASH_ADD_C
		attr_accessor :hash,:value
		def initialize(hash,value)
			@hash = hash
			@value = value
		end
		def eval
			@hash.eval.merge!(value.eval)
		end
end
####################### HASH REMOVE ##############################
```

```
class HASH_REMOVE_C
      attr_accessor :hash,:index
      def initialize(hash,index)
            @hash = hash
            @index = index
      end
      def eval
            @hash.eval.delete(index.eval)
      end
end
####################### EMPTY ##############################
class EMPTY_C
      attr_accessor :data
      def initialize(data)
            @data = data
      end
      def eval
            BOOL_C.new(@data.eval.empty?).eval
      end
end
```

TTTClasses.rb
Beskriver klasserna för språkets olika konstruktioner.

```
#!/usr/bin/env ruby
#Made in ruby version 1.8.7
@@variables =[{}]
# List with hash that holds the variables in the program.
# The list grows one hash with every scope.
# Stored [{"varibel"=>value,"varibel"=>value},{"varibel"=>value},{}]

@@functions = {}
# HASH that holds the functions that is created in the program.
# Stored {"name_on_function"=>function_class_object}

@@scope = 0
# Global scope that change the scope so local and global scope works.

@@scope_base = [0]
# Basic scope for function so variabel outside the function is
# not reach and variables in the function is local with in it self.

@@Debug = false
# Do Debug set true. false for regulary use

###################### Look up ##############################

def look_up(variable)
      i = @@scope
      while(i>=@@scope_base.last)
      if @@variables[i][variable] != nil
                  return @@variables[i][variable][0]
                  puts "#{@@variables[i][variable]} is found" if (@@Debug)
                  #return var[0]
      end
      i -= 1
      end
      puts "Variable '#{variable}' does not exist."
      :FALSE
end
def look_up_function(function_name)
      if @@functions[function_name] != nil
            return @@functions[function_name]
      end
      puts "Function '#{function_name}' does not exist."
      :FALSE
end
###################### Check type ##############################

def convert_to_object(expr_in)
      #to make type check we need to convert the expr to type object
      if (expr_in.eval.kind_of?Float)
            object= NUM_C.new(expr_in.eval)
      elsif(expr_in.eval.kind_of?String)
            object= STR_C.new(expr_in.eval)
```

```
      elsif(expr_in.eval == :FALSE)
            object= BOOL_C.new(expr_in.eval)
      elsif(expr_in.eval == :TRUE)
            object= BOOL_C.new(expr_in.eval)
      end
      return object
end
####################### declare variable ##############################
def declare_variable(variable,expr,type)
      i = @@scope_base.last
    add_variable = true
    while(i<=@@scope)
      if @@variables[i][variable.name] != nil
                  puts "#{@variable.name} is changed in #{i}" if (@@Debug)
                  @@variables[i][variable.name] =  [expr.eval,type]
                  add_variable = false
            end
      i+=1
    end
    if add_variable
      puts "#{@variable.name} is added in #{@@scope}" if (@@Debug)
      @@variables[@@scope][@variable.name] = [expr.eval,type]
    end
end

####################### SCOPE CONTROL ##############################

def new_scope()
      puts "#{@@scope} is incresed by one" if (@@Debug)
      @@scope +=1
      @@variables << {}
end

def new_scope_base()
      new_scope()
      @@scope_base << @@scope
      puts "#{@@scope_base} is base" if (@@Debug)
end

def close_scope()
      @@variables.pop
      puts "#{@@scope} is decrease by one" if (@@Debug)
      @@scope-=1
      if @@scope < 0
      raise("Scope is less then 0. Error in scope.")
      end
end

def close_scope_base()
      close_scope()
      @@scope_base.pop
      puts "#{@@scope_base} is base" if (@@Debug)
end

####################### FUNCTION ##############################
```

```
class DECL_FUNCTION_C
      attr_accessor :function_name, :parameters, :function_body
      def initialize(function_name, parameters ,function_body)
      @function_name = function_name
      if parameters
            @parameters = parameters
      else
            @parameters = []
      end
      @function_body = function_body
      end
      def eval
            puts "#{function_name} is made" if (@@Debug)
            @@functions[function_name] = [@parameters,@function_body]
            :TRUE
      end
end
class FUNCTION_CALL_C
      attr_accessor :function_name, :arguments
      def initialize(function_name, arguments)
      @function_name = function_name
      @arguments = []
      @arguments = arguments if arguments
      end
      def declare_num_parameter(param,param_counter,arg,arg_counter)
            DECL_C.new(VAR_C.new(param[param_counter]), NUM_C.new(arg
[arg_counter]),param[param_counter-1]).eval
      end
      def declare_str_parameter(param,param_counter,arg,arg_counter)
            DECL_C.new(VAR_C.new(param[param_counter]), STR_C.new(arg
[arg_counter]),param[param_counter-1]).eval
      end
      def declare_bool_parameter(param,param_counter,arg,arg_counter)
            DECL_C.new(VAR_C.new(param[param_counter]), BOOL_C.new (arg
[arg_counter]),param[param_counter-1]).eval
      end
      def run_parameters(parameters,argument)
            number_of_parameters = parameters.length-1
            if (number_of_parameters > 0)
#The to counters for parameters and arguments.
#The parameter has two value for each argument.
#So parameters_countern increase by two each loop and the arguments_counter
increase by one.
                  parameter_counter = 1
                  arguments_counter = 0
                  #Loop and set the parameters to it's value.
                  while(parameter_counter<=number_of_parameters)
                        if parameters[parameter_counter-1] == :NUM
  declare_num_parameter(parameters,parameter_counter,argument,arguments_counter)
                        elsif parameters[parameter_counter-1] == :STR
  declare_str_parameter(parameters,parameter_counter,argument,arguments_counter)
                        elsif parameters[parameter_counter-1] == :BOOL
 declare_bool_parameter(parameters,parameter_counter,argument,arguments_counter)
                        else #THIS IS THE ALL TYPE
                              if @argument[arguments_counter].class == NUM_C
```

```
declare_num_parameter(parameters,parameter_counter,argument,arguments_counter)
                              elsif @argument[arguments_counter].class == STR_C
  declare_str_parameter(parameters,parameter_counter,argument,arguments_counter)
                              elsif @argument[arguments_counter].class == BOOL_C
 declare_bool_parameter(parameters,parameter_counter,argument,arguments_counter)
                              end
                        end
                        parameter_counter+=2
                        arguments_counter+=1
                  end
            end
      end
      def check_parameters(parameters,arguments)
            number_of_parameters = parameters.length-1
      # this is check the quantity is the same for parameters and arguments
            if (parameters.length/2) != (arguments.length)
                  return false
            end
            # this is the check for right type
            if (number_of_parameters > 0)
                  parameter_counter = 1
                  arguments_counter = 0
                  while(parameter_counter<=number_of_parameters)
                        if @arguments[arguments_counter].class == NUM_C
                              if  parameters[parameter_counter-1] != :NUM
                                    return false
                              end
                        elsif @arguments[arguments_counter].class == STR_C
                              if  parameters[parameter_counter-1] != :STR
                                    return false
                              end
                        elsif @arguments[arguments_counter].class == BOOL_C
                              if  parameters[parameter_counter-1] != :BOOL
                                    return false
                              end
                        end
                        parameter_counter+=2
                        arguments_counter+=1
                  end
            end
            return true
      end

      def eval
            argument_list=[]
            #We run eval on all the arguments that is set.
            if (!@arguments.empty?)
                  (0...@arguments.length).each do |i|
                        argument_list[i] = @arguments[i].eval
                  end
            end
            new_scope_base()
            # This is to get the function with parameters and body.
```

```
            function = look_up_function(@function_name)

            # We need to check the parameters are right we do so by running
check_parameters
            if (check_parameters(function[0],argument_list))
                  run_parameters(function[0],argument_list)
                  puts "#{@function_name} is called" if (@@Debug)
                  block = function[1]
                  # Here the body or block in the function is run
                  result = block.eval
                  close_scope_base()
                  return result
            else
                  puts "Argument and parameter error, check quantity or type"
                  return :FALSE
            end
      end
end
####################### STATEMENT LIST ##############################

class STMT_LIST_C
  attr_accessor :stmt, :stmt_list
  def initialize (stmt,stmt_list)
    @stmt = stmt
    @stmt_list = stmt_list
  end
  def eval()
    return_value = @stmt.eval
    if @stmt.class != DONE_C
            @stmt_list.eval
    else
      return return_value
    end
  end
end

####################### DECL ##############################
class DECL_C
      attr_accessor :variable, :expr, :type
      def initialize(var, expr,type)
      @variable = var
      @expr = expr
      @type = type
      end
      def eval
            expr = convert_to_object(@expr)
            if((@type == :ALL) or (@type == expr.type))
            declare_variable(@variable,expr,@type)
      else
            puts "Wrong type definition"
            return :FALSE
      end
      @expr.eval
      end
end
```

```
###################### ASSIGN ##############################
class ASSIGN_C
      attr_accessor :variable, :expr
      def initialize(id, expression)
      @variable = id
      @expr = expression
      end
      def eval
            expr = convert_to_object(@expr)
            i = @@scope_base.last
            found = false

      while(i<=@@scope)
                  if @@variables[i][@variable.name] != nil
                        type = (@@variables[i][@variable.name])[1]
                        if (type == :ALL)
                              return_value = @@variables[i][@variable.name] =
[expr.eval,type]
                              found = true
                        elsif(type == expr.type)
                              return_value = @@variables[i][@variable.name] =
[expr.eval,type]
                              found = true
                        end
                  end
                  i+=1
      end
      if (found == false)
            puts "Wrong type definition"
            return :FALSE
      end
            return_value[0]
      end
end
###################### VARIABLE ##############################
class VAR_C
  attr_accessor :name
  def initialize(variable)
    @name = variable
  end
  def eval
    return look_up(@name)
  end
end
###################### DECL LIST ##############################
class DECL_LIST_C
      attr_accessor :variable, :expr, :type
      def initialize(id, expression,type)
      @variable = id
      @expr = expression
      @type = type
      end
      def eval
            value_type = :NUM if (@expr.eval[0].kind_of?Float)
            value_type = :STR if (@expr.eval[0].kind_of?String)
```

```
            type_error = false
            if(value_type and @type != :ALL)
                  @expr.eval.each do |i|
                        if i.class != @expr.eval[0].class
                              type_error = true
                        end
                  end
            end
            if (type_error)
                  puts "Wrong type definition"
            return :FALSE
            end
            if (@type == :ALL) or (@type == value_type)
                  declare_variable(@variable,@expr,@type)
      else
            puts "Wrong type definition"
            return :FALSE
      end
      @expr.eval
      end
end
####################### DECL HASH ##############################
class DECL_HASH_C
      attr_accessor :variable, :expr, :type
      def initialize(id, expression,type)
      @variable = id
      @expr = expression
      @type = type
      end
      def eval
            declare_variable(@variable,@expr,@type)
      @expr.eval
      end
end

####################### MATH ##############################
class ARIT_OBJECT
      attr_accessor :value1,:value2,:operator
      def initialize (operator,value1,value2)
      @operator = operator
      @value1 = value1
            @value2 = value2
      end

      def eval()
            return instance_eval("#{@value1.eval()} #{@operator}
                        #{@value2.eval()}")
      end
end

class LOG_OBJECT
      attr_accessor :value1,:value2,:operator
      def initialize (operator,value1,value2)
      @operator = operator
      @value1 = value1
```

```
		@value2 = value2
		end

		def eval()
		if @operator == 'AND'
				@operator = 'and'
		elsif @operator == 'OR'
			@operator = 'or'
		end
		return BOOL_C.new(instance_eval("#{@value1.eval()} #{@operator}
				#{@value2.eval()}")).eval
		end
end
class LOG_OBJECT_NOT
		attr_accessor :value1,:operator
		def initialize (operator,value1)
		@operator = operator
		@value1 = value1
		end
		def eval()
		if @operator == 'NOT'
				@operator = 'not'
		end
		return BOOL_C.new(instance_eval("#{@operator} #{@value1.eval()}")).eval
		end
end
class COMP_OBJECT
		attr_accessor :value1,:value2,:operator
		def initialize (operator,value1,value2)
		@operator = operator
		@value1 = value1
		@value2 = value2
		end
		def eval()
		return BOOL_C.new(instance_eval("#{@value1.eval()} #{@operator}
				#{@value2.eval()}")).eval
		end
end

####################### RETURN ##############################
class DONE_C
		attr_accessor :value
		def initialize (value)
			@value = value
		end
		def eval()
			return @value.eval
		end
end
####################### INPUT AND OUTPUT #########################
class PRINT_C
		attr_accessor :value
		def initialize (value)
			@value = value
		end
```

```
	def eval()
		puts "== "+ @value.eval.to_s
	end
end
class READ_C
	attr_accessor :name ,:type
	def initialize (name,type= :STR)
		@name = name
		@type = type
	end
	def eval()
		print "<< "
		if (@type == :NUM)
			indata = NUM_C.new(gets.to_i)
		else
			indata = STR_C.new(gets.to_s)
		end
		return DECL_C.new(VAR_C.new(@name),indata,@type).eval
	end
end
####################### ITERATORS ##############################

class WHILE_C
  attr_accessor :condition, :statement
  def initialize(cond, stmt)
    @condition = cond
    @statement = stmt
  end
  def eval
    new_scope()
    while @condition.eval == :TRUE do
		if @statement.class == DONE_C
			return_value = @statement.eval
			close_scope()
			return return_value
			break
	else
			@statement.eval
			puts "#{@statement.eval} is eval" if (@@Debug)
		end
    end
    close_scope()
    :TRUE
  end
end

class FOR_C
  attr_accessor :iter_var, :start, :range, :increase, :block
  def initialize (iter_var, start, range, increase, block)
    @iter_var = iter_var
    @start = start
    @range = range
    @increase = increase
    @block = block
  end
```

```
  def eval()
      new_scope()
      # Declare the iterator variable to it start value
    DECL_C.new(VAR_C.new(@iter_var[1]), @start, @iter_var[0]).eval
    while (@start.eval <= @range.eval) do
      if @block.class == DONE_C
                  return_value = @block.eval
                  close_scope()
                  return return_value
                  break
      else
                  @block.eval
            end
            # Redeclare the iterator variable in the loop
      DECL_C.new(VAR_C.new(@iter_var[1]), @start = NUM_C.new((@start.eval +
@increase.eval)), @iter_var[0]).eval
    end
    close_scope()
    :TRUE
  end
end

class EACH_C
  def initialize(iter_var, iter_values, block)
            @iter_var = iter_var
            @iter_values = iter_values
            @block = block
      end
      def eval
            new_scope()
            @iter_values.eval.each do |i|
                  if (i.kind_of?Float)
                        DECL_C.new(VAR_C.new(@iter_var[1]), NUM_C.new(i),
@iter_var[0]).eval
                  elsif(i.kind_of?String)
                        DECL_C.new(VAR_C.new(@iter_var[1]), STR_C.new(i),
@iter_var[0]).eval
                  elsif(i == :FALSE)
                        DECL_C.new(VAR_C.new(@iter_var[1]), BOOL_C.new(i),
@iter_var[0]).eval
                  elsif(i == :TRUE)
                        DECL_C.new(VAR_C.new(@iter_var[1]), BOOL_C.new(i),
@iter_var[0]).eval
                  end
                  @block.eval
            end
            close_scope()
            :TRUE
      end
end

####################### SELECT ################################
class IF_C
  attr_accessor :condition,:stmt
  def initialize (condition,stmt)
```

```
    @condition = condition
    @stmt = stmt
  end
  def eval()
      new_scope()
    if @condition.eval()== :TRUE
            return_value = @stmt.eval()
            close_scope()
            return return_value
    end
    close_scope()
  end
end

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

parser.rb
Beskriver lexikalisk analys och parsning med rdparser.

```
#!/usr/bin/env ruby
require './rdparser.rb'
#Made in ruby version 1.8.7

class TTT
  def initialize
    @TTTParser = Parser.new("TTT") do
      token(/\s+/)
      token(/\t+/)
      token(/#.*\/#/)
      token(/TRUE/) {|m| m }
      token(/STR/) {|m| m}
      token(/NUM/) {|m| m}
      token(/FALSE/) {|m| m }
      token(/\IF/) {|m| :IF }
      token(/\ELSE/) {|m| :ELSE }
      token(/\/IF/) {|m| m }
      token(/FOR/) {|m| :FOR }
      token(/LIST/) {|m| m }
      token(/\/LIST/) {|m| m }
      token(/\/FOR/) {|m| m }
      token(/ADD/) {|m| m }
      token(/EMPTY/) {|m|m}
      token(/WHILE/) {|m| :WHILE }
      token(/\/WHILE/) {|m| m}
      token(/EACH/) {|m| m }
      token(/\/EACH/) {|m| m }
      token(/PRINT/) {|m| m }
      token(/\/PRINT/) {|m| m }
      token(/DONE/) {|m|m}
      token(/\/DONE/) {|m|m }
      token(/DECL/){|m| m }
      token(/\/DECL/){|m| m }
      token(/INCBY/){|m|m }
      token(/IN/) {|m|m }
      token(/TO/){|m|m}
      token(/FUNCTION/) {|m| :FUNCTION }
      token(/\/FUNCTION/) {|m|m }
      token(/READ/) {|m| m }
      token(/\/READ/) {|m|m }
      token(/HASH/) {|m|m}
      token(/\/HASH/) {|m|m }
      token(/"[\w\s!\?]*"/) {|m| m.to_s }
      token(/>>/){|m|m}
      token(/==/) {|m| m }
      token(/=\/=/) {|m| m }
      token(/>/) {|m| m }
      token(/>=/) {|m| m }
      token(/</) {|m| m }
      token(/<=/) {|m| m }
      token(/AND/) {|m| m }
```

```
token(/OR/) {|m| m }
token(/NOT/) {|m| m }
token(/\&\&/) {|m| m }
token(/\|\|/) {|m| m }
token(/\d+\.*\d*/) {|m| m.to_f}
token(/\w+/) {|m| m }
token(/./){|m| m }

start :PROGRAM do
  match(:STMT_LIST){|m| m.eval unless m.class == nil }
end
#STMT_LIST
rule :STMT_LIST do
  match(:STMT,:STMT_LIST) {|stmt,stmt_list|
      STMT_LIST_C.new(stmt,stmt_list)}
  match(:STMT)
end
#STMT
rule :STMT do
  match(:RETURN_STMT)
  match(:IO_STMT)
  match(:SEL_STMT)
  match(:ITER_STMT)
  match(:DATA_STMT)
  match(:ASSIGN_STMT)
  match(:FUNCTION_STMT)
  match(:EXPR)
end

rule :FUNCTION_STMT do
  match(:FUNCTION_DEF)
  match(:FUNCTION_CALL)
end

rule :FUNCTION_DEF do
  match(:FUNCTION,:FUNC_NAME,'(',')',:STMT_LIST,'/FUNCTION')
      {|_,name,_,_,body,_| DECL_FUNCTION_C.new(name,nil,body)}
match(:FUNCTION,:FUNC_NAME,'(',:PARAMETER_LIST,')',:STMT_LIST,'/
FUNCTION') {|_,name,_,parameter,_,body,_|
DECL_FUNCTION_C.new(name,parameter.flatten,body)}
end

rule :FUNCTION_CALL do
  match(:FUNC_NAME,'(',')') {|name,_,_| FUNCTION_CALL_C.new(name,nil)}
  match(:FUNC_NAME,'(',:ARGUMENT_LIST,')')
      {|name,_,arg_list,_| FUNCTION_CALL_C.new(name,arg_list.flatten)}
end

rule :ARGUMENT_LIST do
  match(:STMT){|m| [m] }
  match(:ARGUMENT_LIST,',',:STMT){|m,_,n| [m]<<[n] }
end

rule :PARAMETER_LIST do
  match('<',:TYPES,'>',:VAR_DEC){|_,m,_,n| [m,n] }
```

```
    match(:PARAMETER_LIST,',',:PARAMETER_LIST){|m,_,n| [m]<<[n] }
  end

  rule :SEL_STMT do
    match(:IF,'(',:EXPR,')',:STMT_LIST,:ELSE,:STMT_LIST,'/IF')
        {|_,_,cond,_,ifbody,_,elsebody,_|
        IF_ELSE_C.new(cond,ifbody,elsebody)}
    match(:IF,'(',:EXPR,')',:STMT_LIST,:ELSE,:ELSEIF,'/IF')
        {|_,_,cond,_,ifbody,_,elsebody,_|
        IF_ELSE_C.new(cond,ifbody,elsebody)}
    match(:IF,'(',:EXPR,')',:STMT_LIST,'/IF')
        {|_,_,cond,_,ifbody,_| IF_C.new(cond,ifbody)}
  end
  rule :ELSEIF do
    match(:IF,'(',:EXPR,')',:STMT_LIST,:ELSE,:ELSEIF)
        {|_,_,cond,_,ifbody,_,elsebody|
         IF_ELSE_C.new(cond,ifbody,elsebody)}
    match(:IF,'(',:EXPR,')',:STMT_LIST,:ELSE,:STMT_LIST)
        {|_,_,cond,_,ifbody,_,elsebody|
        IF_ELSE_C.new(cond,ifbody,elsebody)}
    match(:IF,'(',:EXPR,')',:STMT_LIST)
        {|_,_,cond,_,ifbody| IF_C.new(cond,ifbody)}
  end

  rule :ITER_STMT do
    match(:WHILE,'(',:EXPR,')',:STMT_LIST,'/WHILE')
        {|_,_,cond,_,body,_| WHILE_C.new(cond,body)}

match(:FOR,'(',:ITER_VAR,'IN',:NUM,'TO',:NUM,'INCBY',:NUM,')',:STMT_LIST,'/FOR')
  {|_, _, iter_var, _, start, _, range, _, increase, _, block, _| FOR_C.new
        (iter_var, start, range, increase, block)}
    match('EACH','(',:ITER_VAR,'IN',:VAR_DEC,')',:STMT_LIST,'/EACH')
        {|_,_,var,_,list,_,body,_| EACH_C.new(var,VAR_C.new(list),body)}
  end

  rule :ITER_VAR do
    match('<',:TYPES,'>',:VAR_DEC) {|_, m, _, n|[m,n]}
  end

  rule :ASSIGN_STMT do
    match('DECL',:ASSIGN_TYPE,:VAR_DEC,':',:EXPR,'/DECL')
        {|_,type,name,_,expr,_| DECL_C.new(VAR_C.new(name),expr,type)}
    match('DECL',:ASSIGN_TYPE,:VAR_DEC,':','/DECL')
        {|_,type,name,_,_| DECL_C.new(VAR_C.new(name),nil,type)}
    match('LIST',:ASSIGN_TYPE,:VAR_DEC,':',:LIST,'/LIST')
        {|_,type,name,_,expr,_| DECL_LIST_C.new(VAR_C.new(name),expr,type)}
    match('HASH',:VAR_DEC,':',:HASH,'/HASH')
        {|_,name,_,expr,_| DECL_LIST_C.new(VAR_C.new(name),expr)}
    match(:VAR_DEC,':',:EXPR){|name,_,expr|
        ASSIGN_C.new(VAR_C.new(name),expr)}
  end
```

```
rule :DATA_STMT do
  match('ADD',:VAR_DEC, :LIST){|_,var,list|
      LIST_ADD_C.new(VAR_C.new(var),list)}
  match('ADD',:VAR_DEC, :HASH){|_,var,hash|
      HASH_ADD_C.new(VAR_C.new(var),hash)}
  match('REMOVE',:VAR_DEC,'[',:NUM,']'){|_,var,_,index,_|
      LIST_REMOVE_C.new(VAR_C.new(var),index)}
  match('REMOVE',:VAR_DEC,'{',:STR,'}'){|_,var,_,index,_|
      HASH_REMOVE_C.new(VAR_C.new(var),index)}
end

rule :ASSIGN_TYPE do
  match('<',:TYPES,'>'){|_,m,_| m}
end

rule :IO_STMT do
  match(:PRINT_STMT) {|m|m}
  match(:READ_STMT) {|m|m}
end

rule :PRINT_STMT do
  match('PRINT',:STMT_LIST,'/PRINT'){|_,m,_| PRINT_C.new(m) }
end

rule :READ_STMT do
  match('READ',:VAR_DEC,'/READ') {|_,m,_| READ_C.new(m) }
  match('READ','<',:TYPES,'>',:VAR_DEC,'/READ')
      {|_,_,t,_,m,_| READ_C.new(m,t) }
end

rule :RETURN_STMT do
  match('DONE',:STMT_LIST,'/DONE') {|_,m,_| DONE_C.new(m) }
end

#EXPR
rule :EXPR do
  match(:EXPR,:OPERATOR_A,:TERM) {|e,o,t| ARIT_OBJECT.new(o,e,t) }
  match(:TERM){|m|m}
end

rule :TERM do
  match(:TERM,:OPERATOR_B,:LOG) {|e,o,t| ARIT_OBJECT.new(o,e,t)}
  match(:LOG){|m|m}
end

rule :LOG do
  match(:LOG,:LOG_OPERATOR,:COMP){|e,o,t| LOG_OBJECT.new(o,e,t) }
  match(:LOG_OPERATOR_NOT,:COMP){|o,t| LOG_OBJECT_NOT.new(o,t) }
  match(:COMP){|m|m}
end

rule :COMP do
  match(:COMP,:COMP_OPERATOR,:FACTOR){|e,o,t| COMP_OBJECT.new(o,e,t) }
  match(:FACTOR){|m|m}
end
```

```
rule :FACTOR do
  match(:FUNCTION_CALL)
  match('(',:EXPR,')') {|_,m,_| m }
  match(:DATA)
  match(:TYPE)
  match(:VAR_CALL)
  match(:LIST)
  match(:HASH)
end
rule :DATA do
  match(:VAR_DEC,'[',:NUM,']')
      {|var,_,num,_| LIST_GET.new(VAR_C.new(var),num)}
  match(:VAR_DEC,'[',:VAR_CALL,']')
      {|var,_,num,_| LIST_GET.new(VAR_C.new(var),num)}
  match(:VAR_DEC,'{',:STR,'}')
       {|var,_,str,_| HASH_GET.new(VAR_C.new(var),str)}
  match(:VAR_DEC,'{',:VAR_CALL,'}')
       {|var,_,str,_| HASH_GET.new(VAR_C.new(var),str)}
end

rule :LIST do
  match('[',:TYPE_LIST,']'){|_,list,_| LIST_C.new(list.flatten) }
  match('[',']'){|_,_| LIST_C.new([]) }
end

rule :TYPE_LIST do
  match(:TYPE) {|m|[m.eval] }
  match(:TYPE_LIST,',',:TYPE) {|m,_,n| [m]+[n.eval] }
end
rule :HASH do
  match('{',:TYPE_HASH,'}'){|_,hash,_| HASH_C.new(hash) }
  match('{','}'){|_,_| HASH_C.new({}) }
end
rule :TYPE_HASH do
  match(:STR,'>>',:TYPE) {|m,_,n| {m.eval=>n.eval} }
  match(:TYPE_HASH,',',:TYPE_HASH) {|m,_,n| m.merge!(n)}
end

rule :COMP_OPERATOR do
  match('==') {|m| m }
  match('=/=') {|m| '!=' }
  match('>')  {|m| m }
  match('>=') {|m| m }
  match('<')  {|m| m }
  match('<=') {|m| m }
end

rule :LOG_OPERATOR do
  match('&&')  {|m| m }
  match('||')  {|m| m }
  match('AND') {|m| m }
  match('OR')  {|m| m }

end
rule :LOG_OPERATOR_NOT do
```

```
      match('!')   {|m| m }
      match('NOT') {|m| m }
    end

    rule :OPERATOR_A do
      match('+') {|m| m }
      match('-') {|m| m }
    end
    rule :OPERATOR_B do
      match('*') {|m| m }
      match('/') {|m| m }
      match('%') {|m| m }
    end
    rule :TYPES do
      match(/NUM/){|m| m = :NUM}
      match(/STR/){|m| m = :STR}
      match(/BOOL/){|m| m = :BOOL}
      match(/ALL/){|m| m = :ALL}
    end
    rule :TYPE do
      match(:NUM)
      match(:STR)
      match(:BOOL)
      match(:ALL)
    end

    rule :NUM do
      match('-',Float){|_,m| NUM_C.new(-m) }
      match(Float){|m| NUM_C.new(m) }
    end

    #STR
    rule :STR  do
      match(/("[\w\s!\?]*")/){|m| STR_C.new(m) }
    end
    #VAR_CALL
    rule :VAR_CALL do
      match(/^[A-Z][a-zA_Z0-9_]*/){|m| VAR_C.new(m)}
    end
    #VAR_DEC
    rule :VAR_DEC do
      match(/^[A-Z][a-zA_Z0-9_]*/){|m| m}
    end
    #FUNC_NAME
    rule :FUNC_NAME do
      match(/[A-Z_]+/){|m|m}
    end
    #BOOL
    rule :BOOL do
      match(/TRUE/) {|m| BOOL_C.new(m) }
      match(/FALSE/){|m| BOOL_C.new(m) }
    end
  end # end of Parser.new
 end # initialize
end
```

rdparser.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

  Match = Struct.new :pattern, :block

  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,:lex_tokens, :current_rule

  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
```