Programkod

Initiate.rb

```
require_relative 'grammar'

if ARGV.size == 0
  puts "Error: no rino-script entered."
elsif File.exist?(ARGV[0])
  content = File.read(ARGV[0])
  RINORules.new(content, ARGV[1..ARGV.size])
else
  puts "Error: file "+ "'" + ARGV[0] + "'" + " was not found."
End
```

Grammar.rb

```
require_relative 'rdparse'
require_relative 'nodes'


class RINORules
  attr_reader :content
  attr_accessor :RINOParser
  def initialize(content, argv)
    @content = content
    @RINOParser = Parser.new("RINOParser") do
      token(/".*"/) {|m| m}

token(/^(?!return$|continue$|break$|if$|elif$|else$|for$|loop$|while$|task$|out$|or$|and$|is$|not$)[a-zA-Z\_]\w*/) { |m| m }
      token(/^\d+\.\d+/) {|m| m.to_f}
      token(/^\d+/) {|m| m.to_i}
      token(/^-\d+\.\d+/) {|m| m.to_f}
      token(/^-\d+/) {|m| m.to_i}
```

```
    #Compare operators:
    token(/<=/) {|m| m}
    token(/>=/) {|m| m}
    token(/==/) {|m| m}
    token(/!=/) {|m| m}

    #Logical operators:
    token(/&&/) {|m| m}
    token(/\|\|/) {|m| m}

    #Ignore comments, spaces
    token(/#.*/)
    token(/\\\\[\W\w]+\/\//)
    token(/\s+/)

    #Single characters
    token(/./) {|m| m}


    start :program do
      match(:global_statements) {|global_statements| Program.new(global_statements)}
    end


    rule :global_statements do
      match(:global_statements, :global_statement) { |global_statements, global_statement|
global_statements + global_statement}
      match(:global_statement) { |global_statement| global_statement}
    end
```

```
    rule :global_statement do
      match(:expression, ';') {|expression, _| [expression]}
      match(:var_decl) {|var_decl| [var_decl]}
      match(:task_decl) { |task_decl| task_decl}
    end

    rule :block do
      match(:block, :local_statement) { |block, local_statement| block + [local_statement]}
      match(:local_statement) { |local_statement| [local_statement]}
    end

    rule :local_statement do
      match(:condition_chain) { |condition_chain| Condition_chain.new(condition_chain)}
      match(:var_decl)
      match(:expression, ';')
      match(:loop_stmt)
    end

    rule :condition_chain do
      match(:if_stmt, :condition_successors) { |if_stmt, condition_successors| if_stmt +
condition_successors}
      match(:if_stmt) { |if_stmt| if_stmt}
    end

    rule :condition_successors do
      match(:elif_stmt, :condition_successors) { |elif_stmt, condition_successors| elif_stmt +
condition_successors}
      match(:elif_stmt) {|elif_stmt| elif_stmt}
      match(:else_stmt) {|else_stmt| else_stmt}
```

```
    end


    rule :if_stmt do
      match("if", '(', :bool_expr, ')', '{', :block, '}') {|_,_,bool_expr,_,_,block,_|
[Condition_stmt.new(block, Expression.new(bool_expr))] }
    end


    rule :elif_stmt do
      match("elif", '(', :bool_expr, ')', '{', :block, '}') {|_, _, bool_expr,_,_,block,_|
[Condition_stmt.new(block, Expression.new(bool_expr))]}
    end


    rule :else_stmt do
      match("else", '{', :block, '}') {|_,_,block,_| [Condition_stmt.new(block)]}
    end


    rule :loop_stmt do
      match("for", '(', :var_decl, :expression, ';', :expression, ')', '{', :block, '}') { |_, _, var_decl,
iterations, _, step, _ ,_ ,block, _| For_loop.new(var_decl, iterations, step, block)}
      match("while", '(', :bool_expr, ')', '{', :block, '}' ) {|_, _, bool_expr, _, _, block, _|
While_loop.new(Expression.new(bool_expr), block)}
    end



    rule :task_decl do
      match("task", :identifier, '(', :param_list, ')', '{', :block, '}')
{|_,identifier,_,param_list,_,_,block,_| [Task_decl.new(identifier.name, block, param_list)] }
      match("task", :identifier, '(', ')', '{', :block, '}') {|_,identifier,_, _, _,block,_|
[Task_decl.new(identifier.name, block)] }
    end
```

```
    rule :var_decl do
      match(:identifier, '=', :expression, ';')  { |identifier, _, expression, _|
Var_decl.new(identifier.name, expression) }
    end

    rule :identifier do

match(/^(?!return$|if$|elif$|else$|for$|loop$|while$|task$|out$|or$|and$|is$|not$)[a-
zA-Z\_]\w*/) { |identifier| Identifier.new(identifier)}
    end

    rule :param_list do
      match(:param_list, ',', :identifier) { |param_list, _, identifier| param_list
+[identifier.name]}
      match(:identifier) { |identifier| [identifier.name] }
    end

    rule :expression do
      match(:bool_expr)  { |bool_expr| Expression.new(bool_expr)}
      match(:aritm_expr) { |aritm_expr| Expression.new(aritm_expr)}
    end

    rule :literal do
      match(:call_stmt)
      match(:boolean)
      match(:numeric)
      match(:string)
      match(:identifier)
    end

    rule :bool_expr do
```

```
    match(:bool_expr, :or_oper, :bool_term)  {|expr, _, term| expr + ['|'] + term}
    match(:bool_term)  {|term| term }
  end

  rule :bool_term do
    match(:bool_term, :and_oper, :bool_factor) {|term, _, factor| term + ['&'] + factor}
    match(:bool_factor) {|factor| factor }
  end

  rule :bool_factor do
    match('(',:bool_expr,')') { |_, expr, _| [Expression.new(expr)] }
    match(:aritm_expr, :compare_op, :aritm_expr)  { |left_expr, compare_op, right_expr|
[Expression.new(left_expr + [compare_op] + right_expr)]  }
    match(:aritm_expr)
  end

  rule :or_oper do
    match("||")
    match("or")
  end

  rule :and_oper do
    match("&&")
    match("and")
  end
```

```
 rule :compare_op do
       match('>')     { '>' }
       match('<')     { '<' }
       match("==")    { "==" }
       match("is")    { "==" }
       match("!=")    { "!=" }
       match("not")   { "!=" }
       match(">=")    { ">=" }
       match("<=")    { "<=" }
     end


     rule :aritm_expr do
       match(:aritm_expr, '+', :aritm_term) {|expr, oper, term| expr + [oper] + term}
       match(:aritm_expr, :aritm_term) {|expr, term| expr + ['+'] + term}
       match(:aritm_expr, '-', :aritm_term) {|expr, oper, term| expr + [oper] + term}
       match(:aritm_term) {|term| term }
     end


     rule :aritm_term do
       match(:aritm_term, '*', :aritm_factor) {|term, oper, factor| [Expression.new(term +
[oper] + factor)]}
       match(:aritm_term, '/', :aritm_factor) {|term, oper, factor| [Expression.new(term +
[oper] + factor)]}
       match(:aritm_factor) {|factor| factor}
     end
```

```
    rule :aritm_factor do
      match('(', :aritm_expr, ')') { |_, expr, _| [Expression.new(expr)] }
      match(:literal) {|literal| [literal]}
    end

    rule :call_stmt do
      match(:rino_stmt)
      match(:identifier, '(', :arg_list, ')') {|identifier,_,arg_list,_|
Call_stmt.new(identifier.name, Arg_list.new(arg_list))}
      match(:identifier, '(', ')') {|identifier,_, _| Call_stmt.new(identifier.name)}
    end

    rule :rino_stmt do
      match(:IO_stmt)
      match(:debug_stmt)
      #...
    end



rule :IO_stmt do
      match(:out_stmt)
      match(:in_stmt)
    end

    rule :debug_stmt do
      match("expr_test", '(',:debug_expr, ',', :expression,')') {|_,_, debug_expr, _, expression2,
_| Expression_test.new(debug_expr, expression2)}
    end
```

```
#Ganska onödigt stor regel kanske.
    rule :debug_expr do
      match(Integer, '+', :debug_expr) { |l, _, d| l.to_s << '+' << d}
      match(Integer, '-', :debug_expr) { |l, _, d| l.to_s << '-' <<d}
      match(Integer, '*', :debug_expr) { |l, _, d| l.to_s << '*' <<d}
      match(Integer, '/', :debug_expr) { |l, _, d| l.to_s << '/' <<d}
      match(Float, '+', :debug_expr) { |l, _, d| l.to_s << '+'<<d}
      match(Float, '-', :debug_expr) { |l, _, d| l.to_s << '-'<<d}
      match(Float, '*', :debug_expr) { |l, _, d| l.to_s << '*'<<d}
      match(Float, '/', :debug_expr) { |l, _, d| l.to_s << '/' <<d}
      match(Float) {|l| l.to_s}
      match(Integer) {|l| l.to_s}
      match("true") {|b| b.to_s}
      match("false") {|b| b.to_s}
    end

    rule :out_stmt do
      match("out", '(', :expression, ')') { |_,_, expression, _| Out_stmt.new(expression) }
    end

    rule :arg_list do
      match(:arg_list, ',', :expression) { |arg_list, _, expr| arg_list + [expr] }
      match(:expression) { |expr| [expr]}
    end

    rule :numeric do
      match(:integer)
      match(:float)
    end
```

```
    rule :boolean do
      match("true") { Rino_bool.new(true) }
      match("false") { Rino_bool.new(false) }
    end

    rule :string do
      match(/".*"/) { |string| Rino_string.new(string)}
    end

    rule :float do
      match(Float) { |float| Rino_float.new(float.to_f)}
    end

    rule :integer do
      match(Fixnum) {|integer| Rino_int.new(integer.to_i) }
    end
  end
  @RINOParser.parse @content
 end
end
```

Nodes.rb

```
require_relative 'data_handler'
require_relative 'error_thrower'

@@error_thrower = ErrorThrower.new
@@data_handler = DataHandler.new(@@error_thrower)
@@scope_counter = 0

class Program
  def initialize(global_block)
    global_block.each do |statement|
      statement.set_scope([0])
    end
    #Sedan evaluera
    global_block.each do |statement|
      statement.evaluate
    end
  end
end
```

```
class Var_decl
  attr_reader :name, :value
  def initialize(name, expression)
    @name = name
    @expression = expression
  end
  def set_scope(scope)
    @my_scope = scope.dup
    @expression.set_scope @my_scope
  end
  def evaluate
    @value = @expression.evaluate
    @@data_handler.declare_var_in_scope(@name, @value, @my_scope)
  end
end

class Task_decl
  def initialize(name, block, param_list = [])
    @name = name
    @block = block
    @param_list = param_list
  end
  def set_scope(scope)
    scope.push(@@scope_counter)
    @my_scope = scope.dup
    @block.each do |statement|
      statement.set_scope(scope)
    end
    @@scope_counter += 1
```

```
    scope.pop
  end
  def evaluate()
    @@data_handler.try_add_task(@name, @param_list, @block, @my_scope.dup)
  end
end

class Arg_list
  def initialize(arg_list)
    @arg_list = arg_list
  end
  def set_scope(scope)
    @my_scope = scope.dup
    @arg_list.each do |expression|
      expression.set_scope @my_scope
    end
  end
  def evaluate
    evaluated_args = []
    @arg_list.each do |expression|
      evaluated_args.push(expression.evaluate)
    end
    evaluated_args
  end
end

class Call_stmt
  def initialize(task_to_call, arg_list = nil)
    @task_to_call = task_to_call
```

```
    @arg_list = arg_list
  end
  def set_scope(scope)
    @my_scope = scope.dup
    if !@arg_list.nil?
      @arg_list.set_scope @my_scope
    end
  end
  def evaluate
    if !@arg_list.nil?
      @@data_handler.try_call_task(@task_to_call, @arg_list.evaluate)
    else
      @@data_handler.try_call_task(@task_to_call, [])
    end
  end
end
```

```
class Condition_chain
  def initialize(condition_stmts)
    @condition_stmts = condition_stmts
  end
  def set_scope(scope)
    @condition_stmts.each do |condition_stmt|
      condition_stmt.set_scope(scope)
    end
  end
  def evaluate
    @condition_stmts.each do |condition_stmt|
      if condition_stmt.evaluate
        break
      end
    end
  end
end
```

```
class Condition_stmt
  def initialize(block, bool_expr = nil)
    @block = block
    @bool_expr = bool_expr
  end

  def set_scope(scope)
    scope.push(@@scope_counter)
    if @bool_expr != nil
      @bool_expr.set_scope(scope)
    end
    @block.each do |statement|
      statement.set_scope(scope.dup)
    end
    @@scope_counter += 1
    scope.pop
  end

  def evaluate
    if @bool_expr == nil || @bool_expr.evaluate
      @block.each do |statement|
        statement.evaluate
      end
      return true
    end
  end
end
```

```
class Out_stmt
  def initialize(expression)
    @expression = expression
  end
  def set_scope(scope)
    @my_scope = scope.dup
    @expression.set_scope @my_scope
  end
  def evaluate
    print "\n\t#{@expression.evaluate}\n"
  end
end

class Expression_test
  def initialize(expected, expression)
    @expected = expected
    @expression = expression
  end
  def set_scope(scope)
    @my_scope = scope.dup
    @expression.set_scope @my_scope
  end
  def evaluate
    expected =  eval @expected
    value = @expression.evaluate
    if expected == value
      print "\n\tSuccess! The rino expression evaluated to the expected expression #{expected} (left).\n\n"
    else
```

```
      print "\n\tFail! The rino expression did not evaluate to the expected #{expected}
(left)\n\tThe result was #{value}.\n\n"
    end
  end
end



class For_loop
  def initialize(iterator_decl, iterations_expr, step_expr = 1, block)
    @iterator_decl = iterator_decl
    @iterations_expr = iterations_expr
    @step_expr = step_expr
    @block = block
  end

  def set_scope(scope)
    scope.push(@@scope_counter)
    @my_scope = scope.dup
    @iterator_decl.set_scope(@my_scope)
    @iterations_expr.set_scope(@my_scope)
    @block.each do |statement|
      statement.set_scope(scope)
    end
    @@scope_counter += 1
    scope.pop
  end
```

```
def evaluate
    @iterator_decl.evaluate
    @iterator_name = @iterator_decl.name
    @iterator_value = @iterator_decl.value
    @iterations = @iterations_expr.evaluate
    @step = @step_expr.evaluate

    if @step < 1
      @@error_thrower.invalid_step_exception(@step)
    end

    (@iterator_value..@iterations-1).step(@step).each do
      @block.each do |statement|
        statement.evaluate
      end
      @@data_handler.iterate_variable(@iterator_name, @my_scope, @step)
    end
  end
end

class While_loop
  def initialize(bool_expr, block)
    @bool_expr = bool_expr
    @block = block
  end
  def set_scope(scope)
    scope.push(@@scope_counter)
    @my_scope = scope.dup
    @bool_expr.set_scope(@my_scope)
```

```
    @block.each do |statement|
      statement.set_scope(scope)
    end
    @@scope_counter += 1
    scope.pop
  end
  #Medans uttrycket är sant kör loopen.
  def evaluate
    while @bool_expr.evaluate
      @block.each do |statement|
        statement.evaluate
      end
    end
  end
end

class Expression
  def initialize (expressions)
    @expressions = expressions
  end
  def set_scope(scope)
    @my_scope = scope.dup
    @expressions.each do |expression|
      if !expression.is_a? String
        expression.set_scope(@my_scope)
      end
    end
  end
```

```
def evaluate
   expressions = @expressions.dup
   value = expressions.shift.evaluate
   while expressions.count != 0
    operator = expressions.shift
    right_value = expressions.shift.evaluate
    value = value.send(operator, right_value)
   end
   value
 end
end

class Identifier
 attr_reader :name,:value
 def initialize(name, value = nil)
  @name = name
  @value = value
 end
 def set_scope(scope)
  @my_scope = scope.dup
 end
 def evaluate
  @value = @@data_handler.try_find_first_occurance(@name, @my_scope.dup)
 end
end
```

```
class Rino_string
  def initialize(value)
    @value = value
  end
  def set_scope(scope)
    @my_scope = scope.dup
  end
  def evaluate
    @value[1..-2]
  end
end

class Rino_int
  def initialize(value)
    @value = value
  end
  def set_scope(scope)
    @my_scope = scope.dup
  end
  def evaluate
    @value
  end
end
```

```
class Rino_float
  def initialize(value)
    @value = value
  end
  def set_scope(scope)
    @my_scope = scope.dup
  end
  def evaluate
    @value
  end
end

class Rino_bool
  def initialize(value)
    @value = value
  end
  def set_scope(scope)
    @my_scope = scope.dup
  end
  def evaluate
    @value
  end
end
```

Datahandler.rb

```
def debug_print(*msgs)
  print("\n\tDEBUG LOGIC PRINT:")
  msgs.each do |msg|
    print("\n\t#{msg}")
  end
  print("\n\n")
end

class DataHandler
  def initialize(error_thrower)
    @error_thrower = error_thrower
    @tasks = {}
    @variables = {[0] => []}
  End

  def try_call_task(task_to_call, arg_list = [])
    if @tasks.has_key? task_to_call
      candidate = @tasks[task_to_call]
      param_list = candidate[0]
      if param_list.count == arg_list.count
        block = candidate[1]
        task_scope = candidate[2].dup
        for i in 0..param_list.count - 1
          declare_var_in_scope(param_list[i], arg_list[i], task_scope)
        end
        block.each do |statement|
          statement.evaluate
```

```
      end
    else
      @error_thrower.invalid_arg_list_size_exception(caller.task_to_call, item.param_list,
caller.arg_list)
    end
  else
    @error_thrower.task_not_found_exception(task_to_call)
  end
 end

 def try_add_task(name, param_list, block, scope)
  if !@tasks.has_key? name
    @tasks[name] = [param_list]+[block]+[scope]
  else
    @error_thrower.task_name_occupied_exception(name)
  end
 end

 def declare_var_in_scope(name, value, scope)
  vars = get_reachable_vars(scope.dup)
  vars.each do |pair|
    if pair[0] == name
      pair[1] = value
    end
  end
  if @variables.has_key? scope
    @variables[scope].push([name] + [value])
  else
    @variables[scope] = [[name] + [value]]
  end
```

```
end

def get_reachable_vars(scope)
  reachable_vars = []
  while scope != []
    if @variables[scope] == nil
      scope = find_next_populated_scope(scope.dup)
    end
    reachable_vars += @variables[scope]
    scope.pop
  end
  reachable_vars
end

def find_next_populated_scope(scope)
  while @variables[scope] == nil
    scope.pop
  end
  scope
end

def try_find_first_occurance(name, scope)
  while scope != []
    if @variables[scope] == nil
      scope = find_next_populated_scope(scope.dup)
    end
    @variables[scope].each do |pair|
      if pair[0] == name
        return pair[1]
```

```
      end
    end
    scope.pop
  end
  @error_thrower.var_not_found_exception(name)
  nil
 end

 def iterate_variable(name, scope, step)
  var = try_find_first_occurance(name, scope)
  declare_var_in_scope(name, var+=step, scope)
 end
 def draw_scopes
  @variables.each do |key, value|
   tabs = "\t"*key.count
   print("\n#{tabs}Scope #{key}:\n")
   value.each do |var|
    print("#{tabs} #{var}\n")
   end
  end
  print("\n")
 end
end
```

Errorthrower.rb

```
class ErrorThrower

  def task_name_occupied_exception(identifier)

    raise format_error("Multiple definitions of task \"#{identifier}\" found.", "Make sure task names are unique.")

  end


  def var_not_found_exception(identifier)

    raise format_error("Variable \"#{identifier}\" was not found in any reachable scope.", "Make sure \"#{identifier}\" is defined in a reachable scope before referencing it.")

  end


  def task_not_found_exception(identifier)

    raise format_error("Task \"#{identifier}\" was not found in the program.", "Make sure \"#{identifier}\" is defined before calling it.")

  end


  def invalid_arg_list_size_exception(task_to_call, param_list, arg_list)

    raise format_error("The size of argument list #{arg_list} is not the same as that of parameter list #{param_list} in task #{task_to_call}.", "Make sure that the size of #{arg_list} matches that of #{param_list}.")

  end


  def invalid_step_exception(step)

    raise format_error("Invalid step counter found #{step}.", "Use only values above 0.")

  end


  def invalid_terminator_stmt_excepction(found_in, stmt)
```

```
    raise format_error("Invalid step counter found #{step}.", "Use only values above 0.")
  end

  def invalid_step_exception(step)
    raise format_error("Invalid step counter found #{step}.", "Use only values above 0.")
  end

  def format_error(error, solution)
    "\n\n\tOpps! Rino has encountered an
error:\n\n\t\t#{error}\n\n\tSolution:\n\n\t\t#{solution}\n\n"
  end
end
```