Untitled

-Datorspråket som ingen känner till

Projekt: Dataspråk, TDP019
Linköping universitet
Alicia Duong alidu423
Jonathan Karlsson jonka918
Handledare: Anders Haraldsson
12/05/20

Innehållsförteckning

1 Inledning

Datorspråket Untitled är skapat av två studenter på programmet Innovativ programmering vid Linköpings universitet. Språket är influerat av programmeringsspråk såsom Ruby, Python och C++. Det är ett enkelt språk skrivet i Ruby. Nedan kommer språket att representeras och diskuteras.

1.1 Syfte

Rapportens huvudsyfte är att ge läsaren en grundförståelse av de teoretiska specifikationerna kring språket Untitled, samt det praktiska användandet av språket.

1.2 Målgrupp

Rapporten är riktad till medstuderande på programmet för Innovativ programmering vid Linköpings universitet, lärare samt personer med intresse för programmeringsspråk. Då målgruppen är personer med en viss förståelse av teknik kommer det i rapporten att antagas att läsaren har en viss uppfattning om de koncept som diskuteras och de tankegångar som förs.

2 Användarhandledning

Denna del av rapporten ska ge läsaren en grundförståelse för att kunna använda sig av Untitled samt förse läsaren med den information som krävs för att kunna sätta igång att skriva ett eget program.

2.1 Om språket

Untitled är ett imperativt dataspråk, som har vissa influenser av språken C++ och Ruby. Utseendemässigt liknar Untitled C++-syntaxen för for- och while-loppar samt if-satser. Likt C++ avslutas även varje statement med ”;” och den har även liknande typning.
De influenser som språket har från Ruby är att Untitled alltid har ett sanningsvärde; alla positiva heltal, chars och strängar i Untitled är sanna medan resten falskt. Språket har även använt sig av Rubys primitiva datatyper och operationer på dessa.

2.2 Installation av språket

För att kunna använda sig av språket behöver man ha Ruby installerat. Detta kan lätt göras genom terminalen.
För Ubuntu och Debian:

```
sudo apt-get install ruby
```

För Mac os X med pakethanteraren macports :

```
sudo port install ruby
```

Språkets tolk körs sedan med hjälp av kommandot:

```
ruby untitled.rb
```

2.3 Programmering av språket

När allt är installerat och redo att köras är det bara att börja skriva kod. Nedan ges instruktioner för olika syntaxer och egenskaper hos Untitled som kan användas.

2.3.1 Datatyper

Untitled har ett relativt begränsat urval datatyper: int, float, string och bool.

2.3.2 Variabler och tilldelning

De variabler som finns i Untitled består av tecknen A-Z samt a-z och kan vara av godtycklig längd. När man tilldelar variabler ett värde kan det se ut enligt följande:

```
int x = 3;
```

Det går också att ändra värdet på en variabel genom att skriva:

```
x = 4;
```

2.3.3 Operatorer och operationer

För att kunna utföra aritmetiska och logiska operationer krävs följande:
Aritmetiska operationer: +, -, *, % och /.
Logiska operationer: !, ||, och &&.
Relationsoperationer: <, >, <=, ==, != och >=.

Följande operatorer tar två operander, dock med undantag för ”!”, och kan utföras på till exempel följande sätt:

Aritmetiska uttryck:

De aritmetiska uttrycken har parenteser och följer standardprioritet. Vid beräkning prioriteras *, % och / över + och -.

```
1 + 2;
1 + 3 * 3 / 1;
x = 3;
x + 1;
f(x) + 1;
(1 + 2) * 3;
x % 2;
```

Logiska uttryck:

Den logiska operatorn är lite speciell. Untitled har alltid ett sanningsvärde:
Språket har stöd för True och False:

```
True är True
False är False
```

Alla positiva heltal är sanna, även flyttal.

```
2 är True
1 är True
0 är False
```

Även strängar har ett sanningsvärde. Alla tomma strängar är False medan icke-tomma strängar är True. Dvs:

```
”” är False
”hej” är True
```

Detta gör att man kan skriva uttryck enligt följande:

```
(!(a==b)) #Detta körs så länge relationen inte stämmer
((4+40) and ”Lisa”) är True
(0 || 1) är True
(0 && 1) är False
```

Observera att variabler som är odeklarerade kommer ge ett felmeddelande och inte False.

Relationsuttryck:

```
(expr == expr)
(a < b)
(1 > 0.8) #Untitled kan jämföra flyttal med intergers.
```

2.3.4 In- och utmatning

För att skriva ut information till terminalen används kommandot *write*. Ett exempel kan vara:

```
write (<något>;);
```

Det är tillåtet att med *write* skriva ut vad som helst ur programmet som har ett returvärde. Untitled har även en *read*-funktion som ser ut som följer:

```
read();
int HEJ = read();
```

Read läser in från programtolken och returnerar värdet. Det går att fånga upp returvärdet och tilldela det till en variabel.

2.3.5 Program på fil

I Untitled använder man *load* för att läsa in ett program från en fil. Ett stycke exempelkod kan se ut enligt följande:

```
load program.un
```

2.3.6 Funktioner

Datorspråket Untitled stödjer funktioner med typade parametrar och en specificerad returtyp. Funktionen i sig består av ett godtyckligt antal så kallade statements, som beskriver vad funktionen ska utföra.
Exempel på giltiga funktioner kan vara:

```
fun int main (string str){
     <statments här>;
     return 0;
}
```

Något som är värt att notera är att det är tillåtet att deklarera funktioner i funktioner. Detta gör att de kan vara lokala för den funktion de deklarerats i och därför inte åtkomliga i det globala scopet.

2.3.7 For-loopar

For-loopar i Untitled fungerar ungefär som i C++, där man anger ett startvärde, ett slutvärde och ett uttryck för att ändra styrvariabeln. Dessa variabler måste deklareras innanför parenteserna och det är även tillåtet att skriva nästlade loopar eftersom kodblocket innanför måsvingarna är en stmt_list.

```
for(int i = 0; i != x; i = i+1){
    <något här> ;
}
```

Observera att i Untitled kan man avbryta en pågående loop genom break.

2.3.8 While-loopar

Likt for-loopen är utseendet lånat från C++, där while-loppen körs så länge det logiska uttrycket stämmer överens och satserna som ska utföras finns mellan två måsvingar. Exempel som i:

```
while (a<b){
        <något här>;
}
```

2.3.9 If-satser

En if-sats i Untitled kan bestå av två delar; if och else. Likt de flesta språk är if-delen obligatorisk medan else är valbar. En vanlig if-sats kan se ut enligt följande:

```
if(a < b){
     <något här> ;
}
else {
     <annat action> ;
}
```

2.3.10 Scope och skuggningar av variabler

Med hjälp av skuggning kan man skapa variabler i det yttersta kodblocket utan att skriva över variabler som befinner sig högre upp i variabelstacken. Detta kan vara användbart i t.ex. repetitionssatser. I kodexemplet nedan så illustreras detta fenomen.

```
int HEJ = 5;
if (2<3){
      int HEJ = 2;
}
HEJ;
```

På sista raden då HEJ skrivs ut är det variabeln som ligger i bas-scopet. Variabeln inne i if-satsen med samma namn tillhör ett inre kodblock och raderas när if-satsen är slut.

2.3.11 Exempel på program

Nedan följer ett kod-exempel som innefattar *write()*, *read()*, *while* samt funktioner.

```
write("Hej och välkommna, gissa på det hemlila värdet!";);
int A = read();

while (A != 10) {
     write(A;);
     A = read();
}
write("Du gissade rätt, 10 var det hemliga värdet!";);

fun int hej() {
     write("hej!!!!";);
}
hej();
```

3 Systemdokumentation

Detta avsnitt presenterar en mer teknisk bild över hur språket fungerar. Det innefattar hur språket går från den skrivna koden till dess att programmet har körts.

3.1 Beskrivning av språket

Untitled är ett imperativt språk med syntax till stor del lånad från C/C++, till exempel har Untitled statisk typning och semikolon som skiljetecken för satser. Till skillnad från C/C++ körs språket i en interpretator likt Ruby. Alla delar av språket är inte implementerade ännu, det som inte fungerar fullt ut är typcheckning, funktioner, break-statement samt sanningsvärden för siffror och strängar.

3.2 Från källkod till exekvering

När användaren skrivit in de statements som ska köras eller en sökväg till en fil som matats in trycks enter ner. Det första som händer är den lexikaliska analysen vilken tokeniserar den textsträng som skickas till parsern. Vid parsningen som följer efter skapas ett parseträd och klassobjekt skapas för de olika noderna i trädet. Därefter evalueras programmet genom att parseträdet traverseras och varje nod kör sin *eval-metod*. Beräkning sker från trädets yttersta grenar och metodernas returvärde skickas vidare upp i trädet. Programmet returnerar sedan vad den sista satsen returnerar.

3.3 Grammatik

<begin> ::= <stmt_list>

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

<stmt> ::= <function_def>
| <if_else_stmt>
| <if_stmt>
| <increament>
| <while_stmt>
| <print_stmt>
| <function_call>
| <for_stmt>
| <bool_expr>
| 'break'
| <return>
| <asgn>
| <re_asgn>

<if_stmt> ::= 'if' '(' <bool_expr> ')' '{' <stmt_list> '}'

<if_else_stmt> ::= 'if' '(' <bool_expr> ')' '{' <stmt_list> '}' 'else' '{' <stmt_list> '}'

<for_stmt> ::= 'for' '('<type> [A-Za-z]+ '=' <aritm_expr> ';' <bool_expr> ';' [A-Za-z]+ '=' <aritm_expr> ')' '{' <stmt_list> '}'

```
<while_expr> ::= 'while' '(' <bool_expr> ')"{' <stmt_list> '}'

<print_stmt> ::= 'write' '('<expr> ')'

<identifier> ::=[A-Za-z]+

<asgn> ::= <type> [A-Za-z]+ '=' <aritm_uttyck> ';'

<re_asgn> ::= [A-Za-z]+ '=' <aritm_uttyck> ';'

<function_def> ::= 'fun' <type> <identifier> '{' <stmt_list> '}'
        |'fun' <type> <identifier> '(' <param_list> ')'{' <stmt_list> '}'

<function_call> ::= <identifier>'('<arg_list>')'

<param_list> ::= <param>[, <param_list>]?

<arg_list> ::= <expr>
        | <artm_expr>

<rel_op> ::= '=='
        | '<='
        | '>='
        | '!='
        | '<'
        | '>'

<increment> ::= <identifier> '++'
                | <identifier> '--'

<bool_expr> ::= <or_test>

<or_test> ::= <or_test> '||' <and_test>
        | <and_test>

<and_test> ::= <and_test> '&&' <not_test>
        | <not_test>

<not_test> ::= <comparasion>
        | '!' <bool_expr>

<comparison>
    <aritm_expr> <rel_op> <aritm_expr>
    <aritm_expr>
    '(' <bool_expr> ')'
  end

<type> ::= 'bool'
```

```
        | 'int'
        | 'void'
        | 'string'
        | 'float'

<aritm_expr> ::= <aritm_expr> '+' <term>
                | <aritm_expr> '-' <term>
                | <term>

<term> ::= <term> '*' <factor>
                | <term> '/' <factor>
                | <factor>

<factor> ::= '('<aritm_expr>')'
                | <function_call>
                | <identifier>
                | <int>

<param> ::= <type> <identifier>



<return> ::= 'return' [<expr>|<aritm_uttyck>]
```

4 Programkod

Projektet innefattar tre filer med rubykod. *kod.rb* innehåller lexer och regler, *klasser.rb* har klasserna för trädets olika noder samt *rdparse.rb* som innehåller parsern.

4.1 Parser och lexer

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

require 'logger'
require 'rdparse'
require 'klasser'

class DataLanguage

        def initialize


   @languageParser = Parser.new("untitled") do

#<#<#:: TOKENIZER ::#>#>#

        token(/\s+/)
        token(/\/\/.*$/)
        token(/\/\*(?m:.*)\*\//)

        token(/write/) {|m| m}
        token(/for/) {|m| m}
        token(/while/) {|m| m}
        token(/if/) {|m| m}
        token(/string/) {|m| m}
        token(/int/) {|m| m}
        token(/char/) {|m| m}
        token(/return/) {|m| m}
        token(/break/) {|m| m}
        token(/true/) {|m| m}
        token(/bool/) {|m| m}
        token(/false/) {|m| m}
        token(/fun/) {|m| m}
        token(/void/) {|m| m}
        token(/else/) {|m| m}
        token(/read/) {|m| m}
        token(/==/) {|m| m}   #=/
        token(/<=/) {|m| m}   #=/
        token(/>=/) {|m| m}   #=/
        token(/!=/) {|m| m}   #=/
        token(/</) {|m| m}   #=/
```

```
        token(/>/) {|m| m}   #=/
        token(/=/) {|m| m}   #=/
        token(/\+\+/) {|m| m}

        token(/--/) {|m| m}
        token(/\|\|/) {|m| m}
        token(/\&\&/) {|m| m}
        token(/!/) {|m| m}
        token(/\".*\"/){|m| m.to_s}
        token(/[A-Za-z]+/) {|m| m}
        token(/\d*\.\d+/) {|m| m.to_f}
        token(/\d+/) {|m| m.to_i}
        token(/-/) {|m| m}
        token(/./) {|m| m}
        token(/;/)

#<#<#:: PARSER ::#>#>#

        start :begin do
                match(:stmt_list) {|m| m.eval}
        end

        rule :stmt_list do
                match(:stmt_list, :stmt) {|a, b| Stmt_list_c.new(a, b)}
                match(:stmt) {|m| m}#{|m| Return_stmt_c.new(m)}
        end

        rule :stmt do
                match(:function_def)
                match(:if_else_stmt)  {|m| m}
                match(:if_stmt)  {|m| m}
                match(:increment, ';') {|m, _| m}
                match(:while_stmt) {|m| m}
                        #match('break', ';' ) {|a| a}
                        #match(:return, ';')  #{|a, _| Return_stmt_c.new(a)}
                match(:print_stmt, ';') {|m, _| m}
                match(:function_call)
                match(:for_stmt) {|m| m}
                match(:bool_expr, ';') {|m, _| m}
                match(:asgn, ';') {|m, _| m}
                match(:re_asgn, ';') {|m, _| m}
        end

        rule :bool_expr do
                match(:or_test)
        end
```

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

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



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

        rule :comparison do
                match(:aritm_expr, :rel_op, :aritm_expr){ |a,b,c| Logical_node_c.new(a,b,c) }
                match(:aritm_expr)
                match('(', :bool_expr, ')') {|_,a,_| a }
        end

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

        rule :if_stmt do
                match('if', '(', :bool_expr, ')', '{', :stmt_list, '}') {|_, _, a, _, _, b, _| If_node_c.new(a,
b)}
        end

        rule :if_else_stmt do
                match('if', '(', :bool_expr, ')', '{', :stmt_list, '}', 'else', '{', :stmt_list, '}') {|_, _, a, _, _, b,
                _, _, _, c, _| If_else_node_c.new(a, b, c)}
        end

        rule :for_stmt do
                match('for', '(', :type, :identifier, '=', :aritm_expr, ';', :bool_expr, ';', :identifier,
                '=', :aritm_expr, ')', '{', :stmt_list, '}')  {|_, _, type, id, _, value, _, bool_expr, _, id2, _,
                aritm_expr, _, _, stmt_list, _| For_node_c.new(type, id, value, bool_expr, id2,
                aritm_expr, stmt_list)}
                        #match('for', '(', :asgn, ';', :expr, ';', :increment, ')', '{', :stmt_list, '}')
        end
```

```
rule :while_stmt do
        match('while', '(', :bool_expr, ')', '{', :stmt_list, '}') {|_, _, a, _, _, b, _|
while_node_c.new(a, b)}
end

rule :print_stmt do
        match('write', '(', :stmt_list, ')') {|_, _, a, _| Print_stmt_c.new(a)}
end

rule :read_stmt do
        match('read', '(', ')' ){|_, _, _| Read_stmt_c.new()}
end

rule :identifier do
        match(/[A-Za-z]+/) {|m| m } #{|m| Variable_finder_c.new(m)}
end

rule :asgn do
        match(:type, :identifier, '=', :aritm_expr) {|type, id, _, aritm_expr| Variable_c.new(id,
        type, aritm_expr) } # Asgn_stmt_c.new(type, id, aritm_expr)}
        match(:type, :identifier, '=', :string) {|type, id, _, string| Variable_c.new(id, type,
        string) }
end

rule :re_asgn do
        match(:identifier, '=', :aritm_expr) {|id, _, aritm_expr| Variable_reasgn_c.new(id,
        aritm_expr)}
end

rule :function_def do
        match('fun', :type, :identifier,'(', ')', '{', :stmt_list, '}')  {|_, type, id, _, _, _, stmt_list, _|
        Function_def_c.new(type, id, stmt_list)}
                #match('fun', :type, :identifier, '(', :param_list, ')', '{', :stmt_list, '}')
end

rule :function_call do
        match(:identifier, '(', ')') {|id, _, _| Function_call_c.new(id, nil)}
                #match(:identifier, '(', :arg_list, ')') {|id, _, arg_list, _| function_call_c.new(id,
                arg_list.flatten)}
end




#rule :param_list do # kan vara
#       match(<param>[, <param_list>]?)
# end
```

```
rule :arg_list do
        match(:stmt) {|m| [m]}
        match(:stmt, ",", :arg_list) {|m, _, n| [m] << [n]}
end

rule :increment do
        match(:identifier, '++') {|id, _| Variable_reasgn_c.new(id, Add_c.new
(Variable_finder_c.new(id), Integer_c.new(1)))}
        match(:identifier, '--') {|id, _| Variable_reasgn_c.new(id, Sub_c.new
(Variable_finder_c.new(id), Integer_c.new(1)))}
end

#rule :param do
#       match(:type, :identifier)
#end

rule :return do
                # match('return', :expr)
        match('return', :aritm_expr) {|_, aritm_expr| Return_stmt_c.new(aritm_expr)}
        match('return', :string) {|_, string| Return_stmt_c.new(string)}
end

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

rule :aritm_expr do
        match(:aritm_expr, '+',:term) {|a, b, c| Aritm_node_c.new(a, b, c) }
        match(:aritm_expr, '-', :term) {|a, b, c| Aritm_node_c.new(a, b, c) }
        match(:term)
end

rule :term do
        match(:term, '*', :factor) {|a, b, c| Aritm_node_c.new(a, b, c) }
        match(:term, '/', :factor) {|a, b, c| Aritm_node_c.new(a, b, c) }
        match(:term, '%', :factor) {|a, b, c| Aritm_node_c.new(a, b, c) }
        match(:factor)
 end

rule :type do
        match('bool') {|m|m}
        match('void') {|m| m}
        match('int') {|m| m}
        match('string') {|m| m}
        match('float') {|m| m}
end
```

```
        rule :factor do
                match(:read_stmt) {|m| m}
                match('(', :aritm_expr, ')') {|_,a,_| a }
                        #match('-(', :aritm_expr, ')') {|_,a,_| a * -1 }
                match(:function_call)
                match(:string) {|m| m}
                match(:identifier) {|m| Variable_finder_c.new(m)}
                match(Float) {|m| Float_c.new(m)}
                match(Integer) {|m| Integer_c.new(m)}
          end
      end
end

def done(str)
        ["quit","exit","bye",""].include?(str.chomp)
end

def file(str)
        return_val = false
        if (str =~ /load/)
                return_val = true
        end
        return_val
end

def roll
        print "[*+ Untitled +*] "
        str = gets
        if (file(str))
                filename = str.gsub(/load\s*/, "").strip.chomp
                puts filename
                str = ""
                if File.exist? filename
                        File.open(filename, 'r') do |x|
                        temp = x.readlines
                        str = temp.join
                end
                puts ">> #{@languageParser.parse str}"

                else
                        puts "File not funded"
                end
                roll
        elsif done(str) then
                puts "Bye."
        else
                puts ">> #{@languageParser.parse str}"
                roll
```

```
        end
end

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

DataLanguage.new.roll
```

4.2 Noder

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

require 'logger'

@@scope = 0
@@variables = [{}]
@@functions = {}                        # {id: [returtyp, inparametrar, stmt_list]}

@@debug = false

def open_scope
        @@variables << {}    #index symbolizes scope
        @@scope += 1
        puts "scope +1,  scope:#{@@scope}" if @@debug
end


def close_scope
        @@variables.pop
        @@scope -= 1
        #puts "scope -1,  scope:#{@@scope}"
        if @@scope < 0
                raise("You cannot close base scope!")
        end
end


class Integer_c
        def initialize(value)
                @value = value
        end

        def eval()
                return @value
        end

end



class Float_c
        def initialize(value)
                @value = value
        end
```

```
        def eval()
                return @value
        end

end

class String_c
        def initialize(value)
                @value = value.to_s
        end

        def eval()
                return @value
        end

end

class Stmt_list_c
        def initialize(stmt_list, stmt)
                @stmt = stmt
                @stmt_list = stmt_list
        end

        def eval()
                #return_val = @stmt.eval

                if @stmt.class != Return_stmt_c
                        @stmt_list.eval
                else
                        return return_val
                end
                return_val = @stmt.eval
        end
end



class Return_stmt_c
        def initialize(expr)
                @expr = expr
        end

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

```
class Factor_c
	def initialize(value)
		@value = value
	end

	def eval()
		return @value
	end

end


class Variable_c
	def initialize(id, type, expr)
		@id = id
		@type = type
		@expr = expr

	end
	def eval()
		#kolla om typen stämmer
		#puts "scope: #{@@scope} "
		#if @value.class != Read_stmt_c
		value = @expr.eval
		#end
		@@variables[@@scope][@id] = [@type, value]
		return @value.eval
	end
end

class Variable_finder_c
	def initialize(id)
		@id = id
	end

	def eval
		for scope in @@variables.reverse
			if scope[@id]
				return scope[@id][1]
			end
		end
		puts "No variable found with name #{@id}!"
		return nil
	end
end

class Variable_reasgn_c
	def initialize(id, newval)
```

```
		@id = id
		@newval = newval
		#puts "newval: #{@newval.eval}"
	end

	def eval

		for scope in @@variables.reverse
			if scope[@id]
				#puts "reasigning, old val: #{i[@id][1]}"
				scope[@id][1] = @newval.eval
				#puts "reasigning, new val: #{i[@id][1]}"
				return scope[@id][1]
			end
		end
		puts "No variable found with name #{@id}!"
		return nil
	end
end



class Function_call_c
	def initialize(id, arg_list)    #hur köra de statements som finns?
		@id = id
		@arg_list = arg_list
	end

	def eval()
		#kolla om id finns i funktionslistan
		if @@functions[@id]
		#kör funktion
			puts "fun call" if @@debug
			open_scope
				#kolla och lagra argument
				if @@functions[@id][1] == nil
					value = @@functions[@id][2].eval
	#evaluera stmt_list i fun
				end
			close_scope
		else
			puts "Function definition not found"
		end
		return value  #returnera funktionens returvärde.   #problem? om void?
	end

end
```

```
class Function_def_c
	def initialize(type, id, stmt_list)
		@type = type
		@id = id
		@stmt_list = stmt_list
	end

	def eval()
		puts "fun def" if @@debug
		@@functions[@id] = [@type, nil, @stmt_list]
		return true
	end

end

class Function_def_param_c
	def initialize(type, id, param_list, stmt_list)
		@value = value
	end

	def eval()
		return @value
	end

end


class Logical_node_c
	attr_accessor :operand_a, :operator, :operand_b

	def initialize(operand_a, operator, operand_b)
		@operand_a = operand_a
		@operator = operator
		@operand_b = operand_b
	end

	def eval
		return instance_eval("#{@operand_a.eval} #{@operator} #{@operand_b.eval}")
	end

end

class If_node_c
  def initialize(cond, stmts)
    @cond = cond
    @stmts = stmts
  end
  def eval
```

```
        open_scope
    if @cond.eval
      return_value = @stmts.eval
      close_scope
      return return_value
    end

    close_scope

  end
end

class If_else_node_c
  def initialize(cond, stmts, else_stmts)
    @cond = cond
    @stmts = stmts
    @else_stmts = else_stmts
  end
  def eval
        open_scope
    if @cond.eval
        return_value = @stmts.eval

    else
        return_value = @else_stmts.eval
    end

    close_scope
    return return_value

  end
end

class While_node_c
  def initialize(cond, stmts)
    @cond = cond
    @stmts = stmts
  end
  def eval
        open_scope
    while @cond.eval do
        #puts "Class: #{@stmts}"
        #puts "V: #{@@variables}"
      @stmts.eval
    end
    close_scope
  end
end
```

```
class Print_stmt_c
        def initialize(stmt)
   @stmts = stmt
  end
  def eval
    printer = @stmts.eval
    puts printer
    return printer
  end
end

class Read_stmt_c
        def initialize()
        end
        def eval
                print "<< "
                @read = gets
                return @read.chomp!
        end
end


class For_node_c
  def initialize(type, id, value, bool_expr, id2, aritm_expr, stmt_list)
        @type = type
        @id = id
        @value = value
   @cond = bool_expr
   @id2 = id2
   @aritm_expr = aritm_expr
   @stmts = stmt_list
  end
  def eval
        open_scope
        Variable_c.new(@id, @type, @value).eval
        while @cond.eval do
                @stmts.eval
                Variable_reasgn_c.new(@id2, @aritm_expr).eval
        end
        close_scope
  end
end


class Not_node_c

        def initialize(operand)
```

```
		@operand = operand
	end

	def eval
		return (not @operand.eval)
	end
end

class Aritm_node_c
	attr_accessor :operand_a, :operator, :operand_b

	def initialize(operand_a, operator, operand_b)
		@operand_a = operand_a
		@operator = operator
		@operand_b = operand_b
	end

	def eval
		return instance_eval("#{@operand_a.eval} #{@operator} #{@operand_b.eval}")
	end

end
```

5 Reflektion

Projektet i sin helhet har varit mycket intressant och lärorikt att utföra. Den främsta utmaningen har varit att försöka översätta en grundidé till ett praktiskt programmeringsspråk. Resultatmässigt hade vi önskat att vi hunnit göra så mycket mer, till exempel hade vi velat göra språket objektorienterat och fått med mer av den funktionalitet som vi hade med som idéer i början. Trots allt är vi i det stora hela är vi ändå nöjda med resultatet.

En ytterligare grundtanke för att försöka skapa ett "bra" och användbart programmeringsspråk är att man ser till språkets användingsområde och dess målgrupp. Språket får inte gå på användarens nerver och det krävs att det är lätt att förstå och enkelt att använda.

Untitleds syntax är riktad mot en van användare och är väldigt likt c++. Vi valde att försöka skapa ett språk där syntaxen är lättläst, genom att alla statements avslutas med ";" och alla loopar och if-satser ramas in med måsvingar. Vi försökte även att göra språket lätthanterlig genom att göra det imperativt och interpreterat språk.

Vid implementeringen av språket har vi stött på några få problem såsom att språket enbart returnerade resultatet av första stmt:en ur en stmt-list samt att read-funktionen la till gömda nyrads-tecken. Men i det stora hela har processen genom projektet varit lindrig.