LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Språk Dokumentation; Baljan-Language

Author

Daniel Sundmark, dansu239@student.liu.se
Viktor Löfstrand, viklo003@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

VT 2024
Version 1.0

2024-04-30

Innehåll

1 Inledning

Detta är dokumentationen för programmeringsspråket Baljan-Language. Detta projekt påbörjades under våren 2024, inom kursen TDP019, på IP-programmet år 1. Kursen TDP019 handlar om att få en djupare förståelse för implementeringen och designen av programmeringsspråk genom att man skapar sitt eget programmeringspråk. Syftet med dokumentet är att ge en tydlig förståelse för vårt programmeringsspråk och att beskriva hur man använder och vidareutvecklar vårt språk.

1.1 Introduktion

Vi har valt att skapa ett objektorienterat språk. Språket kombinerar olika delar från programmeringspråken; Ruby och C++. Språket är ett generellt programmeringspråk som stödjer bland annat loopar, variabler, vilkorsatser och funktioner.

2 Användarhandledning

Information om hur du installerar språket, vad du behöver för att använda det, och hur du kör språket.

2.1 Installation

För att använda vårt språk Baljan, måste man först ladda ner vår källkod från Git och därefter uppdatera till den senaste versionen av Ruby. Därefter, behöver man öppna terminalen och navigera till katalogen ”programs”. I ”programs” kan du skapa en ny fil med ”bl” ändelse för att skriva kod. För att kompilera koden i terminalen, så skriver man ”run.rb” och namnet på den fil man har skapat.

1. Ladda ner källkoden
2. Uppdatera till den senaste versionen av Ruby

2.2 Användning

När du har installerat språket så kan du nu börja använda det. För att börja så behöver du skapa en fil med filändelsen *.bl* där du kan skriva ditt program. För de olika konstruktionerna språket stödjer, och hur de används, kan du titta på i avsnittet Konstruktioner. När du har skrivit ett program så vill du använda runbl.rb för att kompilera och köra koden.

```
$ ./runbl.rb <flaggor> <program>
```

Program byter du ut mot din fil som har filändelsen .bl och du kan använda vilken kombination du vill av nedanstående flaggor.

-debug Debug utskrifter för parsern

-tree Skriver ut hela programmet i en trädstruktur

-norun Stänger av körning av programmet

2.3 Konstruktioner

Alla olika konstruktioner språket stödjer och hur man använder dem.

2.3.1 Datatyper

Här är de olika datatyper som vårt språk stödjer:

int Heltal

float Flyttal

bool Sanningsvärde

list<datatype> Datastruktur för att lagra en samling data. ”datatype” är den datatyp som listan innehåller.

2.3.2 Variabler

Variabler i språket kan deklareras på följande sätt:

datatyp namn Variabel initiering; *datatyp* som inte är en datastruktur, *namn* som börjar på en bokstav

datatyp<extra> namn Samma som den föregående, men med *extra* som används av datastrukturer

namn *namn* som börjar på en bokstav; användning av en initierad variabel

variablel = värde *variabel* är antingen en variabel initiering eller en redan initierad variabel. Det går att ha flera tilldelningar efter varandra på samma rad för att tilldela flera variabler samtidigt.

```
int x;
list<float> l;
x = 5;
x = int y = 7
```

2.3.3 Print

Print används för att skriva ut värden till terminalen i Baljan.

print *värde* Skriver ut *värde* ifall värde stödjer utskrift

print Skriver ut en tom rad

```
int x = 5;
print x;
print;
print 123;
-> 5
->
->123
```

2.4 Uttryck

Varje uttryck måste avslutas med ett semikolon (;).

```
int x = 5;
print x;
return x;
```

2.5 Loopar

Loopar används för att upprepa en viss kodsekvens flera gånger baserat på ett eller flera villkor.

for(*initiering*; *villkor*; *steg*) *initiering* kallas innan loopen börjar, kan användas för att initiera en indexvariabel. *villkor* kontrolleras varje gång loopen ska köras, villkoret måste evaluera till ett sanningsvärde. *steg* kallas i slutet av varje varv av loopen, kan användas för att stega fram en indexvariabel.

while(*villkor*) *villkor* kontrolleras varje gång loopen ska köras, villkoret måste evaluera till ett sanningsvärde.

break Avbryter loopen oavsett om villkoret fortfarande är sant eller inte.

continue Hoppar över resten av koden i loopen och går till slutet av det nuvarande varvet.

En loop behöver också en kropp definierad direkt efter sig för att veta vilken kod som tillhör loopen.

```
for (int y = 0; y < 5; y = y+1)
{
    if (y == 3) begin
        print 1337;
    end
}
```

```
int x = 0;
while ( x < 5 )
{
    x = x+1 + (x * 2 / 3);
    print x;
}
```

2.6 Vilkorssatser

Vilkorssatser används för att kolla om ett villkor är falskt eller sant och utföra olika handlingar beroende på resultatet.

if(*villkor*) *villkor* bestämmer om koden efter if-satsen ska evalueras, villkoret måste evaluera till ett sanningsvärde.

else if(*villkor*) *villkor* bestämmer om koden efter elseif-satsen ska evalueras, villkoret måste evaluera till ett sanningsvärde. En elseif evalueras enbart om den föregående villkorssatsen evaluerade till false.

else Måste existera efter en annan villkorssats, kommer alltid att evaluera koden efter sig om alla tidigare villkorssatser evaluerade till false.

En villkorssats behöver också en kropp definierad direkt efter sig för att veta vilken kod som tillhör villkorssatsen.

```
if(y < 5)
{
    return foo(y + 1) + y;
}
else
    {
        return (y);
    }
}
```

2.7 Operatorer

Baljan har 3 olika slags typer av operatorer; Artimetiska operatorer, Logiska operatorer och Jämförelseoperatorer. Språket innehåller också matematiska prioriteringar. De aritmetiska uttrycket som man vill prioritera omges med en parentes.
Nedan är alla operatorer i fallande prioritet

1. %
2. * /
3. + -
4. >= > == != <= <
5. && || !

Exempel på prioritering av operatorer:

```
(3-1) * (1+1)
-> 4
```

2.8 Funktioner

Funktioner består av 3 delar: Returtypen, namnet på funktionen och parametrarna. Den första delen av funktionen är returtypen; efter returtypen skrivs sedan namnet på funktion. Därefter skrivs parametrarna för funktionen inom parenteser, särskilda av kommatecken. Sist så definieras kroppen av funktionen där koden som funktionen ska evalueras skrivs.
Varje funktion måste också inkludera en ”return” som säger vad funktionen ska returnera. Om funktionen når slutet utan att den evaluerar en ”return” så kommer programmet att ge ett error. En return definieras genom nyckelordet *return* följt av ett uttryck som evalueras och sedan returneras från funktionen.

```
int main()
begin
    int x = 5;
    print x;
    if(foo(1) == 15)
    {
        print 9001;
        break;
    }
    print 10000;

    return x - 3;
end
```

2.9 Listor

Listor används för att spara en mängd data i en variabel.

list[*index*] Används för att hämta eller sätta värdet för ett redan skapat element. *index* är ett uttryck som måste evaluera till ett heltal.

list.insert(*index*, *värde*) Lägger till ett nytt element på "*index*" i en lista. *index* är ett uttryck som måste evaluera till ett heltal. *värde* är ett uttryck som måste evaluera till den datatyp listan innehåller.

list.remove(*index*) Tar bort ett element vid "*index*" från en lista. *index* är ett uttryck som måste evaluera till ett heltal.

list.count() Returnerar hur många element listan innehåller.

Exempel:

```
list<int> lista;    -> []
lista.insert(0, 5); -> [5]
lista.insert(1, 3); -> [5, 3]
lista[0] = 1;       -> [1, 3]
lista.remove(0);    -> [3]
lista.count();      -> 1
```

3 Systemdokumentation

3.1 Grammatik för språket

Språkets grammatik beskrivs i BNF. Denna inkluderar regler för bland annat våra operatorer, utryck, satser, funktioner med mera. Vi hade inte tid att implementera klasser eller strängar så även fast de är med i BNFen så går de inte att använda i språket.

3.1.1 BNF

```
<program> ::= <defs>
<defs> ::= <def> <defs>
         | <def>

<def> ::= <funcDef>
        | <classDef>
        | <assignment>

<classDef> ::= /class/ <className> <begin> <classBodyDefs> <end>
             | /class/ <className> /</ <className> <begin> <classBodyDefs> <end>

<classBodyDefs> ::= <classBodyDef> <classBodyDefs>
                  | <classBodyDef>

<classBodyDef> ::= <classFuncDef>
                 | <classAssignment>

<funcDef> ::= /int/ /main/ <begin> <statements> <end>
            | <dataType> <funcName> /(/ <funcParamsCreate> /)/ <begin> <statements> <end>
            | <dataType> <funcName> /(\s*)/ <begin> <statements> <end>
            | <dataType> <funcName> <begin> <statements> <end>

<classFuncDef> ::= <className> /initialize/ /(/ <funcParamsCreate> /)/ <begin> <statements> <end>
                 | <accessSpecifier> <dataType> <funcName> /(/ <funcParamsCreate> /)/ <begin> <statemen
                 | <accessSpecifier> <dataType> <funcName> /(\s*)/ <begin> <statements> <end>
                 | <accessSpecifier> <dataType> <funcName> <begin> <statements> <end>

<statements> ::= <statement> <endLine> <statements>
                     | <statement> <endLine>

<statement> ::= <ifStatement>
              | <whileStatement>
              | <forStatement>
              | <break>
              | <continue>
              | <return>
              | <assignment>
              | <print>

<funcCall> ::= /call/ <funcName> /(/ <funcParams> /)/
             | /call/ <funcName> /(\s*)/
```

```
             | /call/ <funcName>

<classFuncCall> ::= /call/ <className> /\./ <funcName> /(/ <funcParams> /)/
                  | /call/ <className> /\./ <funcName> /(\s*)/
                  | /call/ <className> /\./ <funcName>

<funcParams>::= <funcParam> /,/ <funcParams>
              | <funcParam>

<funcParam>::= <expression>



<ifStatement> ::= /if/ /(/ <expression> /)/ <begin> <statements> <end> <elseIfStatement>
                | /if/ /(/ <expression> /)/ <begin> <statements> <end> <elseStatement>
                | /if/ /(/ <expression> /)/ <begin> <statements> <end>

<elseIfStatement> ::= /else/ /if/ /(/ <expression> /)/ <begin> <statements> <end> <elseIfStatement>
                    | /else/ /if/ /(/ <expression> /)/ <begin> <statements> <end> <elseStatement>
                    | /else/ /if/ /(/ <expression> /)/ <begin> <statements> <end>

<elseStatement> ::= /else/ <begin> <statements> <end>

<whileStatement> ::= /while/ /(/ <expression> /)/ <begin> <statements> <end>

<forStatement> ::= /for/ /(/ <assignment> <endLine> <expression> <endLine> <expression> /)/ <begin> <sta
                 | /for/ /(/ <assignment> /in/ <varName> /)/ <begin> <statements> <end>

<break> ::= /break/

<continue>::= /continue/

<return>::= /return/ <expression>
          | /return/

<assignment> ::= <dataType> <varName> /=/ <expression>
               | <varName> /=/ <expression>
               | <dataType> <varName>

<classAssignment> ::= <accessSpecifier> <dataType> <varName> /=/ <expression>
                    | <accessSpecifier> <dataType> <varName>

<getStatement>::= <varName>

<funcParamsCreate> ::= <funcParamCreate> /,/ <funcParamsCreate>
                     | <funcParamCreate>

<funcParamCreate> ::= <dataType> /ref/ <varName>
                    | <dataType> <varName>

<print>::= /print/ <expression>
```

```
<dataType> ::= /int/
             | /float/
             | /string/
             | /bool/
             | /array/
             | /textBox/
             | /button/
             | /size/
             | /point/
             | <className>

<value> ::= <intType>
          | <floatType>
          | <stringType>
          | <boolType>
          | <array>
          | <textBox>
          | <sizeType>
          | <pointType>
          | <button>

<intType> ::= /\d+/
<floatType> ::= /\d+/
              | /\d+\.\d+/
<stringType> ::= /\"[^\"]*\"/
<boolType> ::= /true/
             | /false/

<array> ::= /\[/ <elements> /\]/


<elements> ::= <element> /,/ <elements>
             | <element>

<element> ::= <expression>


<funcName> ::= /[A-Za-z]\w+/
<varName> ::= /[A-Za-z]\w+/
<className> ::= /[A-Za-z]\w+/

<accessSpecifier> ::= /public/
                    | /private/

<begin> ::= /{/
          | /begin/

<end> ::= /}/
        | /end/
```

```
<endLine> ::= /;/

<expression> ::= <boolExpression> <boolLogicOperator> <boolExpression>
               | /!/ <boolExpression>
               | <boolExpression>

<boolLogicOperator> ::= /\|\|/
                      | /\&\&/

<boolExpression> ::= <arithmeticExpression> <comparisonOperator> <arithmeticExpression>
                   | <arithmeticExpression>

<comparisonOperator> ::= /==/
                       | /!=/
                       | /</
                       | />/
                       | /<=/
                       | />=/

<arithmeticExpression> ::= <arithmeticExpression> <arithmeticOperatorA> <term>
                         | <term>

<arithmeticOperatorA> ::= /+/
                        | /-/

<term> ::= <term> <arithmeticOperatorB> <modExpression>
         | <modExpression>

<arithmeticOperatorB> ::= /*/
                        | /\//

<modExpression> ::= <modExpression> <arithmeticOperatorC> <data>
                  | <data>

<arithmeticOperatorC> ::= /%/
                        | /\^/

<data> ::= <funcCall>
         | <getStatement>
         | <value>
         | /(/ <expression> /)/
```

3.2 Delar av systemet

Vårt system består av lexikalisk analys, parsning och evaluering.

1. Lexikalisk analys: I den lexikalisk analysen av skriven programkod görs en skanning för att hitta nyckelord, symboler och andra lexikala element. Vi använder reguljära uttryck för att matcha de lexikaliska elementen och skapa tokens som vi använder vid parsning.
2. Parsning: Vår parser tolkar sekvensen av ”tokens” från den lexikaliska analysen och tillämpar syntax-reglerna vi har skapat för att hitta olika konstruktioner som t.ex. tilldeling eller funktionsdefinitioner

för att skapa ett abstrakt syntaxträd.

3. Evaluering: Vid evalueringen tar vi syntaxträdet som vi skapade vid parsningen och evaluerar det. Alltså under evalueringen utförs de operationer som är definierade i programmet.

3.3 Klasser och relationer

Här beskrivs de viktiga klasserna som används av allt i programmet.

3.3.1 BLScope

BLScope är den klass som håller reda på information om de variabler som existerar i det block den tar hand om. När programmet skapar en variabel så gör den det genom BLScope så att BLScope kan ha koll på all information om variabeln. När vi behöver veta vart värdet av en variabel är sparat eller vilken typ variabeln är så frågar vi BLScope om den infromationen. Varje scope har en signatur för att enkelt veta vart varje variabel skapats så att vi kan ta bort dem snabbt när vi tar bort ett scope. BLScope kan också skapa temporärva variabler som inte har något namn för att kunna passera runt värden mellan noder i programmet.

3.3.2 BLProgram

BLProgram är en förvarings och hjälpklass för programmet. BLProgram innehåller alla funktioner och globala variabler innan programmet körs och när programmet ska köras så lägger BLProgram in alla funktioner i BLFuncionManager och initierar alla variabler i det globala scope;et.

3.3.3 BLRuntime

BLRuntime hjälper till att skapa alla globala variabler programmet behöver och startar exekveringen av programmet genom att kalla på en main funktion som inte tar in några argument. När programmet är färdigt så rensar BLRuntime också upp resterna av programmet och skriver ut vad main returnerade.

3.3.4 BLSignatureManager

BLSignatureManager håller koll på vilka scope signaturer som används i nuläget så att inga scopes ska få samma signatur. När programmet behöver en signature för ett scope så frågar programmet om en signatur och när programmet slutar använda signaturen så säger programmet att det är färdigt med signaturen så den kan användas igen.

3.3.5 BLMemoryManager

BLMemmoryManager är den klass som håller reda på värdet för alla variabler i programmet. När vi vill skapa en ny variabel så säger vi till BLMemmoryManager vilket scope som skapar variabeln och hur många id;n på rad vi vill använda. BLMemmoryManager reserverar då så många id;n på rad som vi vill ha och returnerar det första av dem. Sedan kan ange värden eller hämta värden för varje id. sedan när vi är färdiga med ett id så säger vi till BLMemmoryManager att vi är färdiga med det värdet och då gör vi det möjligt för det id;t att återanvändas.

3.3.6 BLFunctionManager

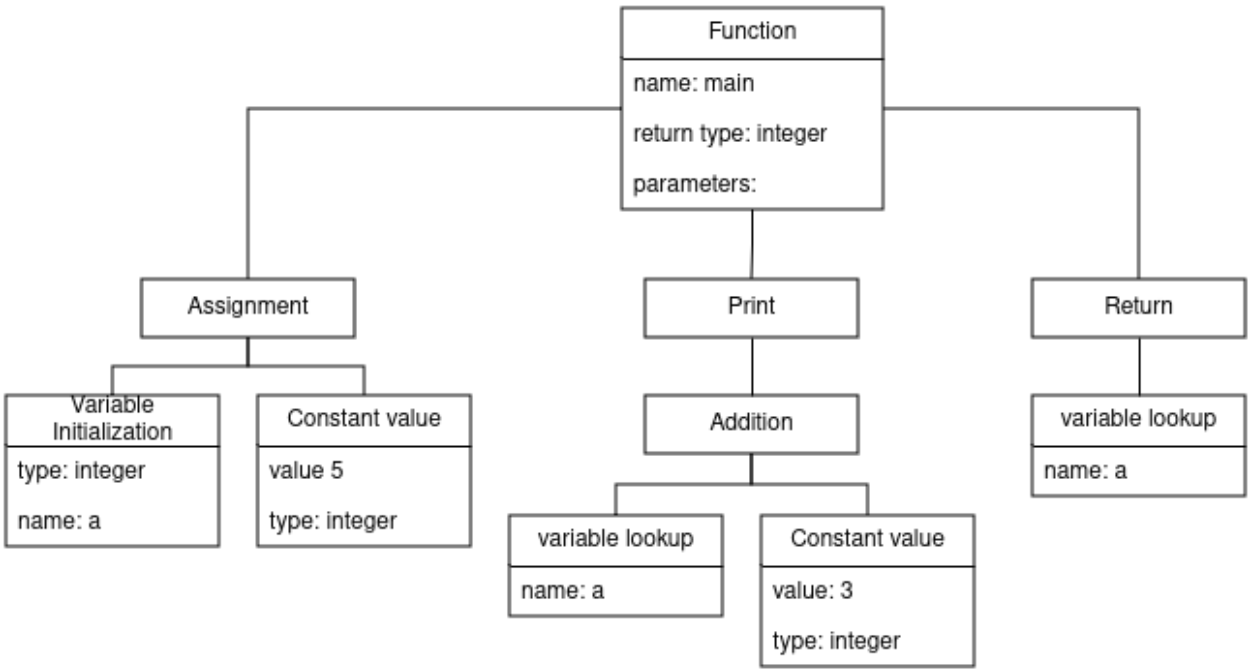
BLFunctionManager hanterar alla funktioner för programmet. Vi kan lägga till funktioner och vi kan hämta funktioner med hjälp av funktionssignaturer. Funktionssignaturer är en databehållare som häller reda på funktionens namn, returtyp, parameter-typer och parameter-namn. BLFunctionManager kan spara och särskilja funktioner med samma namn baserat på deras parametrar så att vi kan definiera flera funktioner med samma namn men med olika parametrar.

3.4 Representation av tokens och syntaxträd

Varje konstruktion i språket har en tillhörande nod som används för att bygga ett abstrakt syntaxträd. Sedan är det det abstrakta syntaxträdet som anänds vi exekveringen av programmet.

```
int main
{
    int a = 5;
    print a + 3;
    return a;
}
```

Bildar syntaxträdet



3.5 Kodstandard

Vi har försökta följa en konsekvent och lättläst kodstandard när vi skrev koden. Koden innehåller tydlig namngivning, indentering och kommentarar som förklara de olika delar av koden.

3.6 Packetering av kod

Projektkoden kan laddas ner från denna gitlab länk: `https://gitlab.liu.se/dansu239/baljanlang`

4 Erfarenheter och reflektion

Innan kursen började hade vi läst en kurs som hette Konstruktion av datorspråk (TDP007) där vi introducerades till, och fick en grundläggande förståelse för, bland annat parsning och tokenisering. Det gav oss en viss erfarenhet av de centrala delarna som krävs för att implementera ett programmeringsspråk. I början av kursen var vår uppgift att själva designa språket, och vi fick hjälp av en handledare.

I vår första språkspecifikation planerade vi att skapa en egen parser. Men senare, i den sista språkspecifikationen, insåg vi att det skulle kräva för mycket arbete att göra detta. Vi valde istället att använda rdparse som vi hade blivit lite bekanta med under TDP007. Vi hade ungefär 2-3 veckor på oss att skapa en språkspecifikation vilket var bra med tanke på att det är utmanande att skapa en fullständig design för ett programmeringsspråk, särskilt med vår begränsade erfarenhet.

Vår slutliga språkspecifikation innebar att vi ville skapa ett generellt språk som kunde hantera bland annat villkorssatser, loopar, listor med mera, med ett särskilt fokus på användargränssnitt (Graphical User Interface, GUI). Vi ville implementera datatyper som ”texbox” och ”button” som enkelt kunde anpassas för det användaren vill göra. Dock har vi ännu inte hunnit implementera detta, eftersom vi har lagt vårt huvudfokus på att skapa en bra grund för ett generellt språket och se till att alla grundläggande funktioner fungerar.

Det vi har lyckats implementera är de grundläggande elementen; variabler, aritmetik, operatorer, listor, villkorssatser, loopar, utskriftssatser, olika datatyper och funktioner.

implementeringen av vissa funktioner, såsom utskriftssatser, while-loopar, datatyper och viss aritmetik, var relativt enkla jämfört med andra delar.

Däremot var det mer svårare att implementera exempelvis for-loopar, hantera listor och definiera hur variabler skulle hanteras. Dessa var mer utmanade eftersom de hade en högre komplexitet. En av de största förbättringar vi har gjort under utvecklingsarbetet var hur variabler ska hanteras. Variabelhanteringen är en central del i vårt språk, så det är viktigt att den utförs korrekt för att vi ska kunna implementera de andra funktionerna på ett effektivt sätt. Bland det sista vi gjorde var att ändra hur variabler fungerade i språket men jag tror att det finns en bättre lösning än den vi implementerade för språket.

Vi borde lagt mera tid på planeringen av projektet för att tidigt kunna identifiera de problem som vi stötte på under projektets gång. om vi hade hittat problemen under planeringen av projektet hade vi kunnat åtgärda dem snabbt och enkelt. Eftersom vi inte hittade dem under planeringen så behövde vi spendera mycket tid på att skriva om stora delar av språket för att lösa de problem vi hade.

5 Programkoden

5.1 nodes.rb

```
require_relative "./variable_nodes.rb"
require_relative "../runtime.rb"

#        Function nodes
#==============================

# The function node that runs a function
class BLFunc
    attr_reader :signature, :body
    def initialize(signature, body)
        @signature = signature
        @body = body
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Setup a variable for the return value to be put in later
        return_addr = scope.add(:return, @signature.return_type)
        $memory.set(return_addr, nil)

        # Some standard variables loops need to function
        break_addr = scope.add(:break, BLBool)
        continue_addr = scope.add(:continue, BLBool)
        $memory.set(break_addr, false)
        $memory.set(continue_addr, false)

        # The loop that evaluates every line of the function, breaks when a returnvalue has been assigne
        @body.each do |stmt|
            stmt.eval(scope)
            scope.clear_temp()
            break if($memory.get(return_addr) != nil)
        end

        # Raise an error if the return value was never set
        if($memory.get(return_addr) == nil)
            raise "No return statement found in function \"#{@signature.name}\""
        end

        # Raise an error if the return value is of the wrong type
        if(scope.type(return_addr) != @signature.return_type)
            raise RuntimeError.new("Value being returned from #{@signature.name} is of wrong type. Suppo
        end

        # Create a temporary variable in the global scope that is used to pass the value back to where t
        return_var = $global.temp(@signature.return_type, $memory.get(return_addr))
        scope.cleanup()
        return return_var
    end
```

```
    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "#{@signature.return_type.type} #{@signature.name}"

        puts "#{level} Args"
        @signature.args.each_with_index do |arg, index|
            if(index + 1 < @signature.args.length)
                print level + "   "
                arg.print_tree(level + "   ")
            else
                print level + "   "
                arg.print_tree(level + "    ")
            end
        end

        puts "#{level} Body"
        @body.each_with_index do |stmt, index|
            if(index + 1 < @body.length)
                print level + "   "
                stmt.print_tree(level + "    ")
            else
                print level + "   "
                stmt.print_tree(level + "    ")
            end
        end
    end
end

class BLFuncCall
    attr_reader :name, :args

    def initialize(name, args)
        @name = name.to_sym
        @args = args
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate all the parameters to find out what type they are
        arguments = []
        @args.each do |arg|
            arguments.append(arg.eval(scope))
        end
        # Get the types of every argument
        arg_types = []
        arguments.each do |arg|
            arg_types.append(scope.type(arg))
        end
        # Make a signature using the name of the function and the types of each argument
        func = $functions.get(BLFuncSignature.new(@name, nil, arg_types))
```

```
        # Setup a new scope for the function
        new_scope = BLScope.new($signatures.get(), $global)

        # Create the parameter variables inside the new scope
        arguments.each_with_index do |arg, index|
            addr = new_scope.add(func.signature.args[index].name, arg_types[index])
            $memory.set(addr, $memory.get(arguments[index]))
        end

        # Run the function and return the address where the return value has been put
        return func.eval(new_scope)
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Func call: #{@name}"
        @args.each_with_index do |arg, index|
            if(index + 1 < @args.length)
                print level + " "
                arg.print_tree(level + "  ")
            else
                print level + " "
                arg.print_tree(level + "   ")
            end
        end

    end
end

class BLReturn
    attr_reader :value
    def initialize(value)
        @value = value
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Get the return address
        return_addr = scope.address(:return)
        value_addr = @value.eval(scope)
        if(scope.type(return_addr) != scope.type(value_addr))
            raise RuntimeError.new("Returning value of wrong type!")
        end
        # Copy the return value into the address of the return variable
        scope.type(return_addr).copy(value_addr, return_addr)
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
```

```
        return "return"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Return"
        print level + " "
        @value.print_tree(level + "  ")
    end
end

#      Signature nodes
#============================

# Signature for identifying functions
class BLFuncSignature
    attr_reader :name, :return_type, :args

    def initialize(name, return_type, args)
        @name, @return_type, @args = name.to_sym, return_type, args
    end


    def eql?(other)
        if( other != nil &&
            other.is_a?(BLFuncSignature) &&
            other.name == @name &&
            @args.length == other.args.length)

            @args.each_with_index do |arg, index|
                # We check if the list contains a func param or not because that makes it so we can che
                type = (arg.is_a?(BLFuncParam) ? arg.type : arg)
                other_type = (other.args[index].is_a?(BLFuncParam) ? other.args[index].type : other.arg
                if(type != other_type)
                    return false
                end
            end

            return true
        end

        return false
    end

    def to_s
        return "Name: #{@name}, Returntype: #{@return_type}, Args: #{@args}"
    end
end

# Signature for identifying member functions
class BLMethodSignature < BLFuncSignature
```

```
    def eql?(other)
        if( other != nil &&
            other.is_a?(BLMethodSignature) &&
            other.name == @name)

            if(self.class <= other.class)
                return true
            end
        end

        return false
    end
end

# Class for storing func parameters
class BLFuncParam
    attr_reader :name, :type
    def initialize(name, type)
        @name = name.to_sym
        @type = type
    end

    # The function that is called when running the program to evaluate each node
    def eval(block)
        raise "Eval should not be called on a BLFuncParam object"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Func param, Name: #{@name}, Type: #{@type}"
    end
end

#        Other nodes
#==============================

class BLPrint
    def initialize(rh = nil)
        @rh = rh
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate the statement that is going to be printed
        out_addr = nil
        if(@rh != nil)
            out_addr = @rh.eval(scope)
            if(out_addr == nil)
                raise "Something went wrong with print, statemet evaluated to nil"
            end
            # Get the type of the return value so we can call its output function
```

```
            type = scope.type(out_addr)
            type.output(out_addr)
        end
        print '\n'
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "print"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Print"
        print level + " "
        @rh.print_tree(level + "  ")
    end
end
```

5.2 operator nodes.rb

```
require_relative "./variable_nodes.rb"


#   Special node for calling member functions
# ============================================

# node that calles a member function
class BLMemberCall
    def initialize(var, name, params)
        @var = var
        @name = name.to_sym
        @params = params
    end

    # Helper function for evaluating the parameters
    def eval_params(scope)
        # Goes through and evaluates all the parameters
        evaled_params = []
        @params.each do |param|
            evaled_params.append(param.eval(scope))
        end
        return evaled_params
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluates all the parameters and gets what type they are
        evaled_params = eval_params(scope)
        param_types = []
```

```
        evaled_params.each do |param|
            param_types.append(scope.type(param))
        end

        # Evaluate the node that gives the address and type of the variable whos member function we wan
        var_addr = @var.eval(scope)
        var_type = scope.type(var_addr)

        # Check if the variable we want to call supports the function we want to call with the paramete
        if(not var_type.implements_method?(BLMethodSignature.new(@name, nil, param_types)))
            raise RuntimeError.new("#{var_type.type} does not implement #{@name}")
        end

        # Call the member function using send
        return var_type.send(@name, var_addr, *evaled_params, scope)
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts  "Member call"
        puts  level + " Base"
        print level +  "   "
        @var.print_tree(level + "    ")
        puts  level + " Member name: #{@name}"
        puts  level + " Parameters"
        @params.each_with_index do |parameter, index|
            if(index + 1 < @params.length)
                print level + "   "
                parameter.print_tree(level + "    ")
            else
                print level + "   "
                parameter.print_tree(level + "    ")
            end
        end
    end
end

#         Common nodes for operators
# ============================================

class BLBinaryOperator
    def initialize(lh, rh)
        @rh = rh
        @lh = lh
    end

    # Common function for evaluating the prameters for the node
    def eval_params(scope)
        left = @lh.eval(scope)
        right = @rh.eval(scope)
        return left, right
```

```
    end

    # Function for printing a tree representation of the program
    # Common function for all binary methods because they all are built similarly
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " Right side"
        print level + "   "
        @rh.print_tree(level + "    ")
        puts level + " Left side"
        level += "  "
        print level + " "
        @lh.print_tree(level + "  ")
    end
end

class BLUnaryOperator
    def initialize(node)
        @node = node
    end

    # Function for printing a tree representation of the program
    # Common function for all unary methods because they all contain the same data
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " Right side"
        level += "  "
        print level + " "
        @node.print_tree(level + "  ")
    end
end

#               Assignment node
# ================================================

class BLAssignment < BLBinaryOperator
    def initialize(lh, rh)
        @lh = lh
        @rh = rh
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate the parameters and get their types
        left, right = eval_params(scope)
        left_type, right_type = scope.type(left), scope.type(right)
        # Check that the variables are the same type
        if(left_type != right_type)
            raise RuntimeError.new("Assignment of wrong type, tried assigning #{right_type} to #{left_ty
        end
        # Copy the value of right into left
```

```
        left_type.copy(right, left)
        scope.remove_temp(right)
        # Return the address of left so assignments can be stacked
        return left
    end
end

#   Unary and binary nodes for the arithmetic
# ============================================

class BLBinaryMethod < BLBinaryOperator
    def initialize(lh, rh, name)
        @lh = lh
        @rh = rh
        @name = name.to_sym
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate the parameters and get their types
        left, right = eval_params(scope)
        left_type, right_type = scope.type(left), scope.type(right)

        # Check if the variable supports the operation
        if(not left_type.implements_method?(BLMethodSignature.new(@name, nil, [left_type, right_type])))
            raise RuntimeError.new("#{left_type.type} does not implement #{@name} with #{right_type.type
        end

        # Call the function to preform the operation
        result = left_type.send(@name, left, right, scope)
        scope.remove_temp(left)
        scope.remove_temp(right)

        # Return the address where the result is stored
        return result
    end

    # Function for printing a tree representation of the program
    # Special function because the node can represent multiple operators
    def print_tree(level = "")
        puts "#{self.class} #{@name}"
        puts level + " Right side"
        print level + "   "
        @rh.print_tree(level + "    ")
        puts level + " Left side"
        level += "  "
        print level + " "
        @lh.print_tree(level + "  ")
    end
end
```

```
class BLUnaryMethod < BLBinaryOperator
    def initialize(lh, name)
        @lh = lh
        @name = name.to_sym
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate the input parameter ang get its type
        left = lh.eval(scope)
        left_type = scope.type(left)

        # Check if the variable supports the given function
        if(not left_type.implements_method?(BLMethodSignature.new(@name, nil, [left_type])))
            raise RuntimeError.new("#{left_type.type} does not implement #{@name}")
        end

        # call the given function
        result = left.send(@name, left, scope)
        scope.remove_temp(left)

        # Return the address where the result is stored
        return result
    end
end

#              Comparison nodes
# ============================================

class BLBinaryComparison < BLBinaryOperator
    # Common function for evaluating the prameters for the node
    # Does comparison using the spaceship operator and returns the result of the comparison
    def eval_comparison(scope)
        left = @lh.eval(scope)
        right = @rh.eval(scope)
        result = $memory.get(left) <=> $memory.get(right)
        if(result == nil)
            raise "Comparison between #{left.class.type} and #{right.class.type} is not possible"
        end
        scope.remove_temp(left)
        scope.remove_temp(right)
        return result
    end
end

class BLEqual < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) == 0)
    end
end
```

```
class BLNotEqual < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) != 0)
    end
end

class BLLessThan < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) == -1)
    end
end

class BLLessThanOrEqual < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) != 1)
    end
end

class BLGreaterThan < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) == 1)
    end
end

class BLGreaterThanOrEqual < BLBinaryComparison
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.temp(BLBool, eval_comparison(scope) != -1)
    end
end

#             Logic operator nodes
# ============================================

class BLBinaryLogicOperator < BLBinaryOperator
    # Common function for evaluating the prameters for the node
    def eval_params(scope)
        # Evaluate the params
        left = @lh.eval(scope)
        right = @rh.eval(scope)

        # Check that both parameters are boolean because this function if for logic operators
        if(scope.type(left) != BLBool  || scope.type(right) != BLBool)
            raise "#{self.class.type} given an argument that is not a bool, argument types were #{scope
        end
```

```
        # Return the addresses where the results are stored
        return left, right
    end
end

class BLUnaryLogicOperator < BLUnaryOperator
    # Common function for evaluating the prameters for the node
    def eval_params(scope)
        # Evaluate the param
        value = @node.eval(scope)

        # Check that the value is a boolean because this is for logic
        if(scope.type(value) != BLBool)
            raise "Cannot preform #{self.class.type} operation on #{scope.type(value).type}, has to be a
        end

        # Return the addresses where the result is stored
        return value
    end
end

class BLLogicAnd < BLBinaryLogicOperator
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        left, right = eval_params(scope)
        result = scope.temp(BLBool, $memory.get(left) && $memory.get(right))
        scope.remove_temp(left)
        scope.remove_temp(right)
        return result
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "And"
    end
end

class BLLogicOr < BLBinaryLogicOperator
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        left, right = eval_params(scope)
        result = scope.temp(BLBool, $memory.get(left) || $memory.get(right))
        scope.remove_temp(left)
        scope.remove_temp(right)
        return result
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "Or"
    end
```

```
end

class BLLogicNot < BLUnaryLogicOperator
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        left = eval_params(scope)
        result = scope.temp(BLBool, !$memory.get(left))
        scope.remove_temp(left)
        return result
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "Not"
    end
end
```

5.3 selector nodes.rb

```
require_relative "./nodes.rb"
require_relative "./variable_nodes.rb"

class BLSelector
    attr_reader :in_loop
    def initialize(selector_stmt, body)
        @selector_stmt = selector_stmt
        @body = body
        @in_loop = false

        # If we are initializing a loop than go through and set all selectors inside to say we are a loo
        # this is used by if and else so they know if break or continue are valid inside them
        if(self.class < BLLoop)
            @body.each do |stmt|
                if(stmt.class < BLSelector)
                    stmt.in_loop = true
                end
            end
        end
    end

    # Helper function to evaluate the selector statements and check if their result is a bool
    def eval_selector(scope)
        result_addr = @selector_stmt.eval(scope)
        if(!(scope.type(result_addr) == BLBool))
            raise "#{self.class.type} was given #{scope.type(result_addr)} as a selector, needs to be a
        end
        return result_addr
    end

    # Assignment function for in_loop that goes through itself in order to tell selectors inside of it t
    def in_loop=(in_loop)
        # We dont go through and set it we are a loop because then we have alredy told all selectors ins
```

```
        if(self.class < BLLoop)
            return nil
        end
        @in_loop = in_loop
        if(in_loop)
            @body.each do |stmt|
                if(stmt.class < BLSelector)
                    stmt.in_loop = true
                end
            end
        end
        return nil
    end


end

# Empty class that is used to identify loops
class BLLoop < BLSelector

end

class BLWhile < BLLoop
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Get the addresses of all the special variables
        break_addr = scope.address(:break)
        continue_addr = scope.address(:continue)
        return_addr = scope.address(:return)

        # while selector_stmt evals to true and the break_flag is false
        while($memory.get(eval_selector(scope)) && (!$memory.get(break_addr)))
            # Loop for evaluating all the statements in the body
            @body.each do |stmt|
                stmt.eval(scope)
                if($memory.get(return_addr) != nil)
                    # If return has been set then set break to true and break out of the body eval
                    $memory.set(break_addr, true)
                    break
                elsif($memory.get(break_addr))
                    # Break out of the body eval if break is set
                    break
                elsif($memory.get(continue_addr))
                    # Set continue to false and break out of the body eval
                    $memory.set(continue_addr, false)
                    break
                end
            end
        end
        # Reset the continue and break variables in case we are in a loop
        $memory.set(break_addr, false)
```

```
        $memory.set(continue_addr, false)
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "while"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " Selector"
        print level + "   "
        @selector_stmt.print_tree(level + "    ")

        puts level + " Body"
        level += "  "
        @body.each_with_index do |stmt, index|
            if(index + 1 < @body.length)
                print level + " "
                stmt.print_tree(level + "  ")
            else
                print level + " "
                stmt.print_tree(level + "  ")
            end
        end
    end
end

class BLFor < BLLoop
    def initialize(var_init, selector_stmt, increment_stmt, body)
        @var_init = var_init
        @selector_stmt = selector_stmt
        @increment_stmt = increment_stmt
        @body = body

        @body.each do |stmt|
            if(stmt.class < BLSelector)
                stmt.in_loop = true
            end
        end
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Evaluate the first statement in the for declaration so we create the loop variable
        @var_init.eval(scope)
        # Get the addresses of all the special variables
        break_addr = scope.address(:break)
        continue_addr = scope.address(:continue)
```

```
        return_addr = scope.address(:return)

        while($memory.get(eval_selector(scope)) && (!$memory.get(break_addr))) #while selector_stmt eval
            @body.each do |stmt|
                stmt.eval(scope)
                if($memory.get(return_addr) != nil)
                    # If return has been set then set break to true and break out of the body eval
                    $memory.set(break_addr, true)
                    break
                elsif($memory.get(break_addr))
                    # Break out of the body eval if break is set
                    break
                elsif($memory.get(continue_addr))
                    # Set continue to false and break out of the body eval
                    $memory.set(continue_addr, false)
                    break
                end
            end
            # Check if break if set so we can exit the while loop before calling the increment statement
            break if($memory.get(break_addr))
            @increment_stmt.eval(scope)
        end
        # Reset the continue and break variables in case we are in a loop
        $memory.set(break_addr, false)
        $memory.set(continue_addr, false)
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "for"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " Var Init"
        print level + "   "
        @var_init.print_tree(level + "    ")
        puts level + " Selector"
        print level + "   "
        @selector_stmt.print_tree(level + "    ")
        puts level + " Increment"
        print level + "   "
        @increment_stmt.print_tree(level + "    ")

        puts "#{level} Body"
        @body.each_with_index do |stmt, index|
            if(index + 1 < @body.length)
                print level + "    "
                stmt.print_tree(level + "    ")
```

```
            else
                print level + "   "
                stmt.print_tree(level + "    ")
            end
        end
    end
end

class BLIf < BLSelector
    @next_selector_stmt = nil
    # Sets the next node for an if elseif else chain
    def set_next_selector_node(selector_node)
        @next_selector_stmt = selector_node
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        selector_value = eval_selector(scope)
        #If selector_stmt evals to true
        if($memory.get(eval_selector(scope)))
            # Get the addresses of all the special variables
            break_addr = scope.address(:break)
            continue_addr = scope.address(:continue)
            return_addr = scope.address(:return)

            @body.each do |stmt|
                stmt.eval(scope)
                break if($memory.get(return_addr) != nil)
                if($memory.get(break_addr))
                    # Break if we are in a loop, otherwise raise error
                    break if(@in_loop)
                    raise SyntaxError.new("Invalid break in if, break can only exist inside loops")
                elsif($memory.get(continue_addr))
                    # Break if we are in a loop, otherwise raise error
                    break if(@in_loop)
                    raise SyntaxError.new("Invalid continue in if, continue can only exist inside loops
                end
            end
        # If selector statement evaluated to false then call the next node in the if, else if, else cha
        elsif(@next_selector_stmt != nil)
            @next_selector_stmt.eval(scope)
        end
        return nil
    end

    # Same as in the base class but we also set the in_loop variable for the next node in the if, else
    def in_loop=(in_loop)
        @in_loop = in_loop
        if(in_loop)
            @body.each do |stmt|
                if(stmt.class < BLSelector)
```

```
                    stmt.in_loop = true
                end
            end
        end
        if(@next_selector_stmt != nil)
            @next_selector_stmt.in_loop = true
        end
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "if"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " In loop: #{@in_loop}"
        puts level + " Selector"
        print level + "   "
        @selector_stmt.print_tree(level + "    ")

        temp_level = level
        if(@next_selector_stmt != nil)
            puts level + " Body"
            temp_level += "  "
        else
            puts level + " Body"
            temp_level += "   "
        end

        @body.each_with_index do |stmt, index|
            if(index + 1 < @body.length)
                print temp_level + " "
                stmt.print_tree(temp_level + "  ")
            else
                print temp_level + " "
                stmt.print_tree(temp_level + "   ")
            end
        end

        if(@next_selector_stmt != nil)
            print level + " "
            @next_selector_stmt.print_tree(level + "  ")
        end
    end
end

class BLElse < BLSelector
    def initialize(body)
```

```
        @body = body
        @in_loop = false
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Get the addresses of all the special variables
        break_addr = scope.address(:break)
        continue_addr = scope.address(:continue)
        return_addr = scope.address(:return)

        @body.each do |stmt|
            stmt.eval(scope)
            break if($memory.get(return_addr) != nil)
            if($memory.get(break_addr))
                # Break if we are in a loop, otherwise raise error
                break if(@in_loop)
                raise SyntaxError.new("Invalid break in if, break can only exist inside loops")
            elsif($memory.get(continue_addr))
                # Break if we are in a loop, otherwise raise error
                break if(@in_loop)
                raise SyntaxError.new("Invalid continue in if, continue can only exist inside loops")
            end
        end
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "else"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "#{self.class}"
        puts level + " In loop: #{@in_loop}"
        puts level + " Body"
        level += "  "
        @body.each_with_index do |stmt, index|
            if(index + 1 < @body.length)
                print level + " "
                stmt.print_tree(level + "  ")
            else
                print level + " "
                stmt.print_tree(level + "  ")
            end
        end
    end
end

class BLBreak
```

```
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Set the break variable to true
        break_addr = scope.address(:break)
        $memory.set(break_addr, true)
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "break"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Break"
    end
end

class BLContinue
    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Set the continue variable to true
        continue_addr = scope.address(:continue)
        $memory.set(continue_addr, true)
        return nil
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "continue"
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Continue"
    end
end
```

5.4 variable nodes.rb

```
require_relative "./nodes.rb"
require_relative "../runtime.rb"

class BLVarInit
    def initialize(type, name, extra = nil)
        @type = type
        @name = name.to_sym
        @extra = extra
    end

    # The function that is called when running the program to evaluate each node
```

```
    def eval(scope)
        addr = scope.add(@name, @type)
        if(@type < BLDataStructure)
            @type.init(addr, scope.signature, @extra)
        else
            @type.init(addr, scope.signature)
        end
        return addr
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        if(@extra != nil)
            puts "Var init: #{@type}<#{@extra}> #{@name}"
        else
            puts "Var init: #{@type} #{@name}"
        end
    end
end

class BLVarNode
    def initialize(name)
        @name = name.to_sym
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        return scope.address(@name)
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Variable: #{@name}"
    end
end

class BLConstValue
    def initialize(value, type)
        @value = value
        @type = type
    end

    # The function that is called when running the program to evaluate each node
    def eval(scope)
        # Creates a temp variable with the correct value and type and then returns its address
        return scope.temp(@type, @value)
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Constant value: #{@type} #{@value}"
```

```
    end
end

# Base class for all varaible types
# The variable classes are never initialized; insted they are just used to make it easier to call the co
class BLVar
    @@method_table = Hash.new

    # Adds a method signature to a variable so other parts of the program knows that the variable suppor
    def self.add_method(method_signature)
        if(not @@method_table.key?(self.name))
            @@method_table[self.name] = Hash.new
        end

        if(@@method_table[self.name].key?(method_signature.name))
            @@method_table[self.name][method_signature.name] += [method_signature]
        else
            @@method_table[self.name][method_signature.name] = [method_signature]
        end
    end

    # Other parts of the program uses this function to ask if a variable supports a method
    def self.implements_method?(method_signature)
        return false if(method_signature == nil)
        return false if(not @@method_table.key?(self.name))
        return false if(not @@method_table[self.name].key?(method_signature.name))

        @@method_table[self.name][method_signature.name].each do |signature|
            return true if(signature.eql?(method_signature))
        end

        return false
    end

    attr_reader :parent_signature
    def self.get(addr)
        raise RuntimeError.new("Base variable class cannot get from memory")
    end
    def self.set(addr, value)
        raise RuntimeError.new("Base variable class cannot set memory")
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "variable"
    end
end

class BLDataStructure < BLVar
    attr_reader :parent_signature
    def self.get(addr)
```

```
        raise RuntimeError.new("#{self.type} cannot get from memory")
    end
    def self.set(addr, value)
        raise RuntimeError.new("#{self.type} cannot set memory")
    end
    def self.copy(addr, to_addr)
        raise RuntimeError.new("#{self.type} cannot copy")
    end
    def self.output(addr)
        raise RuntimeError.new("#{self.type} does not support print")
    end
end



class BLList < BLDataStructure
    # === Initializations ===
    # Reserves memory and sets the necessary variables
    def self.reserve(signature, type)
        addr = $memory.reserve(signature, 4, nil)
        $memory.set(addr, signature)
        $memory.set(addr + 1, type)
        $memory.set(addr + 2, 0)
        return addr
    end

    # Sets som necesarry variables in already reserved memory
    def self.init(addr, signature, extra = nil)
        $memory.set(addr, signature)
        $memory.set(addr + 1, extra)
        $memory.set(addr + 2, 0)
        $memory.set(addr + 3, nil)
        return nil
    end

    # === Internal help functions ===
    # Used to copy a list from one address to another address
    def self.copy(addr, to_addr)
        # Copy the base data in the base of the list
        $memory.set(to_addr + 1, $memory.get(addr + 1))
        $memory.set(to_addr + 2, $memory.get(addr + 2))

        data_type = $memory.get(addr + 1)
        next_node = $memory.get(addr + 3)
        new_next = to_addr + 3
        while(next_node != nil)
            # Reserve enough memory for a node and set the new list to point to that node
            new_addr = $memory.reserve($memory.get(to_addr), data_type.length + 1)

            # Set the addr that has the addr to the next node to the addr of the next node
            $memory.set(new_next, new_addr)
```

```
            new_next = new_addr

            # Copy over the data in the node
            for i in 1...data_type.length + 1
                $memory.set(new_next + i, $memory.get(next_node + i))
            end

            # Move forward to the next node
            next_node = $memory.get(next_node)
        end
        return nil
    end

    # Help function that returns the address of the specified element
    def self.element(addr, index)
        # Get the addr of the first node
        next_element = $memory.get(addr + 3)
        if(next_element == nil)
            raise RuntimeError.new("Index out of range")
        end
        # While index > 0, move to the next node
        while(index > 0)
            next_element = $memory.get(next_element)
            index -= 1
            if(next_element == nil)
                raise RuntimeError.new("Index out of range")
            end
        end

        return $memory.get(next_element)
    end

    # Other parts of the program use this to find out how many memory spaces this class takes up
    def self.length()
        return 4
    end

    # Printing function
    def self.output(addr)
        print '['
        next_element = $memory.get(addr + 3)
        data_type = $memory.get(addr + 1)
        while(next_element != nil)
            data_type.output(next_element + 1)
            next_element = $memory.get(next_element)
            break if(next_element == nil)
            print ', '
        end
        print ']'
    end
```

```
# === Memeber functions the user can access ===
# Returns a refrence to the data at a specified index
def self.index(addr, index_addr, scope)
    if(scope.type(index_addr) != BLInteger)
        raise RuntimeError.new("Can only index using integer")
    end
    index = $memory.get(index_addr)
    # Get the addr of the first node
    next_element = $memory.get(addr + 3)
    if(next_element == nil)
        raise RuntimeError.new("Index out of range")
    end
    # While index > 0, move to the next node
    while(index > 0)
        next_element = $memory.get(next_element)
        index -= 1
        if(next_element == nil)
            raise RuntimeError.new("Index out of range")
        end
    end

    # Create a refrence to the place in memory where the data at the specified index begins
    scope.refrence($memory.get(addr + 1), next_element + 1)
    return next_element + 1
end

# Takes the addr of the list, what index to insert at, and the addr where the value to insert is st
def self.insert(addr, index_addr, value_addr, scope)
    data_type = $memory.get(addr + 1)
    if(data_type != scope.type(value_addr))
        raise "Tried adding wrong type of value to list. Contains #{data_type}, given #{scope.type(
    end
    signature = $memory.get(addr)
    to_assign = nil

    if(scope.type(index_addr) != BLInteger)
        raise RuntimeError.new("Can only index using integer")
    end
    index = $memory.get(index_addr)

    if(index == 0)
        to_assign = addr + 3
    elsif(index == 1)
        to_assign = $memory.get(addr + 3)
    else
        # Use the element function to find the correct element
        to_assign = self.element(addr, index - 2)
    end
    # Creates a new element in the list and initializes it
    # It then copies the value that is going to be stored at that addr from the value addr that was
    old_next = $memory.get(to_assign)
```

```
        new_addr = $memory.reserve(signature, data_type.length + 1)
        $memory.set(new_addr, old_next)
        $memory.set(to_assign, new_addr)
        data_type.init(new_addr + 1, $memory.get(addr))
        data_type.copy(value_addr, new_addr + 1)
        $memory.set(addr + 2, $memory.get(addr + 2) + 1)
        return nil
    end

    # Takes the addr of the list and the index of the element to remove
    def self.remove(addr, index_addr, scope)
        data_type = $memory.get(addr + 1)
        signature = $memory.get(addr)
        if($memory.get(addr + 3) == nil)
            raise RuntimeError.new("Index out of range")
        end
        to_assign = nil

        if(scope.type(index_addr) != BLInteger)
            raise RuntimeError.new("Can only index using integer")
        end
        index = $memory.get(index_addr)

        if(index == 0)
            to_assign = addr + 3
        elsif(index == 1)
            to_assign = $memory.get(addr + 3)
        else
            to_assign = self.element(addr, index - 2)
        end
        to_remove = $memory.get(to_assign)
        new_next = $memory.get(to_remove)
        $memory.release(signature, to_remove, data_type.length + 1)
        $memory.set(to_assign, new_next)
        return nil
    end

    # Get the number of elements in the list
    def self.count(addr, scope)
        return $memory.get(addr + 2)
    end

    # === Extra stuff ===
    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "list"
    end
end
```

```
# Base class for simple variables
# Simple variablesa re variables that only take up one memory address
# Because they all have the same structure, most of their functions are shared from the base class
class BLSimpleVar < BLVar
    # Reserve enough memory for itself and return the address that was reserved for it
    def self.reserve(signature, extra)
        return $memory.reserve(signature, 1)
    end

    # Get teh value at an address
    def self.get(addr)
        return $memory.get(addr)
    end

    # Copy the value form one address to another
    def self.copy(addr, to_addr)
        $memory.set(to_addr, $memory.get(addr))
    end

    # Set the value at an adress
    def self.set(addr, value)
        $memory.set(addr, value)
    end

    # Initialize an address
    def self.init(addr, signature)
        $memory.set(addr, nil)
        return nil
    end

    #Printing function
    def self.output(addr)
        print $memory.get(addr)
    end

    # How many memory slots the variable uses
    def self.length
        return 1
    end
end

class BLBool < BLSimpleVar
    # Special set function to make sure it contains the correct data
    def self.set(addr, value)
        if(value == true || value == false)
            $memory.set(addr, value)
            return nil
        end
        raise RuntimeError.new("Bool assigned value of wrong type")
    end
    # Initialize to base bool value, false
```

```
    def self.init(addr, signature)
        $memory.set(addr, false)
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "bool"
    end
end

# Class for identifying numbers so integers and floats can be used interchangebly where we want to allo
class BLNumber < BLSimpleVar
    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "number"
    end
end

class BLInteger < BLNumber
    # Special set function to make sure it contains the correct data
    def self.set(addr, value)
        if(value.is_a?(Integer))
            $memory.set(addr, value)
            return nil
        end
        if(value.is_a?(Float))
            $memory.set(addr, value.to_i)
            return nil
        end
        raise RuntimeError.new("Integer assigned value of wrong type")
    end

    # Initialize to base integer value, 0
    def self.init(addr, signature)
        $memory.set(addr, 0)
    end

    # All arithmetic functions for integer
    def self.addition(lh, rh, scope)
        return scope.temp(BLInteger, ($memory.get(lh) + $memory.get(rh)).to_i)
    end
    def self.subtraction(lh, rh, scope)
        return scope.temp(BLInteger, ($memory.get(lh) - $memory.get(rh)).to_i)
    end
    def self.multiplication(lh, rh, scope)
        return scope.temp(BLInteger, ($memory.get(lh) * $memory.get(rh)).to_i)
    end
    def self.divition(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh).to_f / $memory.get(rh)).to_f)
    end
    def self.modulo(lh, rh, scope)
```

```
        return scope.temp(BLInteger, ($memory.get(lh) % $memory.get(rh)).to_i)
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "integer"
    end
end



class BLFloat < BLNumber
    # Special set function to make sure it contains the correct data
    def self.set(addr, value)
        if(value.is_a?(Integer))
            $memory.set(addr, value.to_f)
            return nil
        end
        if(value.is_a?(Float))
            $memory.set(addr, value)
            return nil
        end
        raise RuntimeError.new("Float assigned value of wrong type")
    end

    # Initialize to base float value, 0.0
    def self.init(addr, signature)
        $memory.set(addr, 0.0)
    end

    # All arithmetic functions for float
    def self.addition(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh) + $memory.get(rh)).to_f)
    end
    def self.subtraction(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh) - $memory.get(rh)).to_f)
    end
    def self.multiplication(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh) * $memory.get(rh)).to_f)
    end
    def self.divition(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh) / $memory.get(rh)).to_f)
    end
    def self.modulo(lh, rh, scope)
        return scope.temp(BLFloat, ($memory.get(lh) % $memory.get(rh)).to_f)
    end

    # Function for getting a nicer name when printing the tree or when raising errors
    def self.type
        return "float"
    end
```

```
end

BLList.add_method(BLMethodSignature.new("index", BLVar, [BLInteger]))
BLList.add_method(BLMethodSignature.new("insert", nil, [BLInteger, BLVar]))
BLList.add_method(BLMethodSignature.new("remove", nil, [BLInteger]))
BLList.add_method(BLMethodSignature.new("count", BLInteger, []))

BLInteger.add_method(BLMethodSignature.new("addition", BLInteger, [BLInteger, BLNumber]))
BLInteger.add_method(BLMethodSignature.new("subtraction", BLInteger, [BLInteger, BLNumber]))
BLInteger.add_method(BLMethodSignature.new("multiplication", BLInteger, [BLInteger, BLNumber]))
BLInteger.add_method(BLMethodSignature.new("divition", BLInteger, [BLInteger, BLNumber]))
BLInteger.add_method(BLMethodSignature.new("modulo", BLInteger, [BLInteger, BLNumber]))

BLFloat.add_method(BLMethodSignature.new("addition", BLFloat, [BLFloat, BLNumber]))
BLFloat.add_method(BLMethodSignature.new("subtraction", BLFloat, [BLFloat, BLNumber]))
BLFloat.add_method(BLMethodSignature.new("multiplication", BLFloat, [BLFloat, BLNumber]))
BLFloat.add_method(BLMethodSignature.new("divition", BLFloat, [BLFloat, BLNumber]))
BLFloat.add_method(BLMethodSignature.new("modulo", BLFloat, [BLFloat, BLNumber]))
```

5.5 managers.rb

```
# The memory manager
# It keeps trach of all the values of all variables in the program
# It assigns every value a unique address
class BLMemoryManager
    attr_reader :memory, :owners

    def initialize
        @memory = {}
        @owners = {}
    end

    # Reserves memory
    # signature: scope signature to keep track of what scope holds what
    # variables so they can be easily removed when the scope dissapears
    # ammount: is how many consecutive addresses we want to reserve
    # value: is an optional base value to assign to all the variables that gets reserved
    def reserve(signature, ammount, value = nil)
        used = @memory.keys.sort
        addr = nil
        # Special cases for when there are very few variables
        if(used.length() < 1)
            addr = 0
        elsif(used.length() < 2)
            addr = (used[0] > ammount ? 0 : used[0] + 1)
        elsif(used[0] > ammount)
            addr = 0
        else
            # Find the first place where we have enough consecutive addresses free to reserve
            index = 0
            while(index < used.length - 1)
```

```
            if(used[index + 1] - used[index] > ammount)
                addr = used[index] + 1
                break
            end
            index += 1
        end
        if(addr == nil)
            addr = used[used.length - 1] + 1
        end
    end

    # If its the first time the scope reserves variables
    if(!@owners.key?(signature))
        @owners[signature] = []
    end

    # Reserve the addresses and assign their owner
    for i in addr...(addr + ammount) do
        @memory[i] = value
        @owners[signature] += [i]
    end
    # returns the first address of the consecutive addresses reserved
    return addr
end

# Releases already reserved addresses
# signature: the signature of the scope that reserved the addresses
# address: what address to release
# ammount: how many consecutive addresses should be released
def release(signature, address = nil, ammount = nil)
    # If we don't get an address we release all addresses under the specific signature
    if(address == nil)
        if(!@owners.key?(signature))
            return nil
        end
        @owners[signature].each do |addr| @memory.delete(addr) end
        @owners.delete(signature)
        return nil
    end

    # If we don't get an ammount we only release a singe address
    if(ammount == nil)
        @memory.delete(address)
        @owners[signature].delete(address)
        return nil
    end

    # If we get all three parameters then we release all addresses from the address starting point a
    for i in address...(address + ammount) do
        @memory.delete(i)
        @owners[signature].delete(i)
```

```
        end
        return nil
    end

    # Get the value stored at a specified address
    def get(address)
        if(@memory.key?(address))
            return @memory[address]
        elsif(address == nil)
            raise RuntimeError.new("Accessing nil address")
        else
            raise RuntimeError.new("Accessing unreserved memory at address #{address}")
        end
    end

    # Set the value of a specified address
    def set(address, value)
        if(@memory.key?(address))
            @memory[address] = value
        elsif(address == nil)
            raise RuntimeError.new("Accessing nil address")
        else
            raise RuntimeError.new("Accessing unreserved memory at address #{address}")
        end
        return nil
    end

    def to_s
        return "Memory: #{@memory}\nOwners: #{@owners}"
    end
end

# Keeps track of all the functions
class BLFunctionManager
    def initialize
        @funcs = {}
    end

    # Add a function
    def add(func)
        # If there are no functions of this name yet then just add it
        if(!@funcs.key?(func.signature.name))
            @funcs[func.signature.name] = [func]
            return nil
        end

        # If there already are functions of this name then that they don't have the same signatures
        @funcs[func.signature.name].each do |existing_func|
            if(func.signature.eql?(existing_func.signature))
                raise SyntaxError.new("Function \"#{func.signature.name}\" declared more than once with
            end
```

```
        end

        # Check if the function we want to add has the same returntype as other functions of the same na
        if(@funcs[func.signature.name][0].signature.return_type != func.signature.return_type)
            raise SyntaxError.new("Function \"#{func.signature.name}\" has a different return type from
        end

        # If it fufills the requirements we add it
        @funcs[func.signature.name].append(func)
    end

    # Get a function
    def get(signature)
        # If there are no functions with the name given in the signature
        if(not @funcs.key?(signature.name))
            raise RuntimeError.new("No functions matching the given name found.\nSignature: #{signature
        end

        # Find a function with a matching signature
        @funcs[signature.name].each do |func|
            if(func.signature.eql?(signature))
                return func
            end
        end
        # If no function was found then error
        raise RuntimeError.new("No functions matching the given signature found.\nSignature: #{signatur
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Function Manager"
        index1 = 0
        @funcs.each do |key, func_list|
            temp_level = level
            if(index1 + 1 < @funcs.length)
                puts "#{level}#{} #{key}"
                temp_level += "  "
            else
                puts "#{level}#{} #{key}"
                temp_level += "  "
            end

            func_list.each_with_index do |func, index2|
                if(index2 + 1 < func_list.length)
                    print temp_level + " "
                    func.print_tree(temp_level + "  ")
                else
                    print temp_level + " "
                    func.print_tree(temp_level + "  ")
                end
            end
```

```
            index1 += 1
        end
    end
end

# Keeps track of what scope signatures are in use
class BLSignatureManager
    def initialize
        @in_use = []
    end

    # Get a free signature
    def get()
        length = @in_use.length
        if(length < 1)
            @in_use += [1]
            return 1
        elsif(length < 2)
            new_signature = (@in_use[0] == 1 ? 2 : 1)
            @in_use += [new_signature]
            return new_signature
        end
        @in_use.each_with_index do |signature, index|
            if(index > length - 2)
                break
            end
            if(signature + 1 != @in_use[index + 1])
                new_signature = signature + 1
                @in_use += [new_signature]
                return new_signature
            end
        end
        new_signature = @in_use[length - 1] + 1
        @in_use += [new_signature]
        return new_signature
    end

    # Get the signature for global scope
    def get_global()
        if(@in_use.include?(0))
            raise RuntimeError.new("Global scope already in use")
        end
        @in_use += [0]
        return 0
    end

    # Release a signature
    def release(signature)
        @in_use.delete(signature)
        $memory.release(signature)
    end
```

```
end
```

5.6 parser.rb

```
require_relative './rdparse.rb'
require_relative './nodes/variable_nodes.rb'
require_relative './nodes/nodes.rb'
require_relative './nodes/operator_nodes.rb'
require_relative './nodes/selector_nodes.rb'
require_relative './runtime.rb'

class BaljanLang
    attr_accessor :print_tree
    def initialize
        @print_tree = false
        @parser = Parser.new("BaljanLang") do

            token(/\s/)
            token(/\/\*.*\*\//)
            token(/int\b/) {|m| :INTEGER}
            token(/float\b/)  {|m| :FLOAT}
            token(/bool\b/) {|m| :BOOL}
            token(/list\b/) {|m| :LIST}

            token(/==/) {|m|m}
            token(/!=/) {|m|m}
            token(/<=/) {|m|m}
            token(/>=/) {|m|m}
            token(/&&/) {|m|m}
            token(/\|\|/) {|m|m}

            token(/\d+\.\d+/){|m|m}
            token(/\d+/){|m|m}

            token(/[a-zA-Z]\w*/) {|m|m}

            token(/./){|m|m}

            start :program do
                match(:def_list){|list|
                    funcs = []
                    vars = []
                    list.each do |element|
                        if(element.is_a?(BLFunc))
                            funcs.append(element)
                        else
                            vars.append(element)
                        end
                    end
                    BLProgram.new(funcs, vars)
                }
            end
```

```
rule :def_list do
    match(:def, :def_list) {|a, b|  a + b}
    match(:def) {|a| a}
end

rule :def do
    match(:func_def) {|a| [a]}
    match(:variable_init) {|a| [a] }

end

rule :func_def do
    match(:data_type, :name, '(', :func_param_list, ')', :begin, :stmt_list, :end) {|type, n
        BLFunc.new(BLFuncSignature.new(name, type, params), body)
    }
    match(:data_type, :name, '(', ')', :begin, :stmt_list, :end) {|type, name, _, _, _, body
        BLFunc.new(BLFuncSignature.new(name, type, []), body)
    }
    match(:data_type, :name, :begin, :stmt_list, :end) {|type, name, _, body, _|
        BLFunc.new(BLFuncSignature.new(name, type, []), body)
    }
end

rule :variable_init do
    match(:data_type, '<', :data_type, '>', :name) {|type, _, extra, _, name|
        BLVarInit.new(type, name, extra)
    }
    match(:data_type, :name) {|type, name|
        BLVarInit.new(type, name)
    }
end

rule :stmt_list do
    match(:stmt, :stmt_list) {|a, b| a + b}
    match(:stmt){|a| a}
end

rule :stmt do
    match(:if_stmt) {|a| [a]}
    match(:while_stmt) {|a|[a]}
    match(:for_stmt) {|a|[a]}
    match(:break, :end_line) {|a, _|[a]}
    match(:continue, :end_line) {|a, _|[a]}
    match(:return, :end_line) {|a, _|[a]}
    match(:print, :end_line) {|a, _|[a]}
    match(:expression, :end_line) {|a, _|[a]}
end

rule :break do
    match('break') {|_|BLBreak.new()}
```

```
end

rule :continue do
    match('continue') {|_|  BLContinue.new()}
    match('next') {|_|  BLContinue.new()}
end

rule :return do
    match('return', :expression)  {|_, value|  BLReturn.new(value)}
end

rule :print do
    match('print', :expression) {|_, value|  BLPrint.new(value)}
    match('print') {|_|  BLPrint.new()}
end

rule :for_stmt do
    match('for', '(', :expression, :end_line, :expression, :end_line, :expression, ')', :beg
         BLFor.new(var, selector, increment, body)
    }
end

rule :while_stmt do
    match('while', '(', :expression, ')', :begin, :stmt_list, :end) { |_, _, selector, _, _
         BLWhile.new(selector, body)
    }

end

rule :if_stmt do
    match('if', '(', :expression, ')', :begin, :stmt_list, :end, :else_if_stmt){|_, _, sele
        if_stmt = BLIf.new(selector, body)
        if_stmt.set_next_selector_node(next_stmt)
         if_stmt
    }
    match('if', '(', :expression, ')', :begin, :stmt_list, :end, :else_stmt){|_, _, selecto
        if_stmt = BLIf.new(selector, body)
        if_stmt.set_next_selector_node(next_stmt)
         if_stmt
    }
    match('if', '(', :expression, ')', :begin, :stmt_list, :end){|_, _, selector, _, _, bod
         BLIf.new(selector, body)
    }
end

rule :else_if_stmt do
    match('else', 'if', '(', :expression, ')', :begin, :stmt_list, :end, :else_if_stmt){|_,
        if_stmt = BLIf.new(selector, body)
        if_stmt.set_next_selector_node(next_stmt)
         if_stmt
    }
```

```
    match('else', 'if', '(', :expression, ')', :begin, :stmt_list, :end, :else_stmt) {|_, _
        if_stmt = BLIf.new(selector, body)
        if_stmt.set_next_selector_node(next_stmt)
         if_stmt
    }
    match('else', 'if', '(', :expression, ')', :begin, :stmt_list, :end) {|_, _, _, selecto
         BLIf.new(selector, body)
    }
end

rule :else_stmt do
    match('else', :begin, :stmt_list, :end) {|_, _, body, _|
         BLElse.new(body)
    }
end

rule :expression do
    match(:expression, '=', :logic_expression) do |exp1, _, exp2|
        BLAssignment.new(exp1, exp2)
    end
    match(:logic_expression) {|a|a}
end

rule :logic_expression do
    match(:logic_expression, :bool_logic_operator, :bool_expression) {|rh, node, lh|  node.
    match('!', :logic_expression) {|_, value|  BLLogicNot.new(value)}
    match(:bool_expression) {|a|  a}
end

rule :bool_logic_operator do
    match('||'){|_| BLLogicOr}
    match('&&'){|_| BLLogicAnd}
end

rule :bool_expression do
    match(:arithmetic_expression, :comparison_operator, :arithmetic_expression) {|rh, node,
    match(:arithmetic_expression) {|a| a}
end

rule :comparison_operator do
    match('=='){|_| BLEqual}
    match('!='){|_| BLNotEqual}
    match('<'){|_| BLLessThan}
    match('<='){|_| BLLessThanOrEqual}
    match('>'){|_| BLGreaterThan}
    match('>='){|_| BLGreaterThanOrEqual}
end

rule :arithmetic_expression do
    match(:arithmetic_expression, :arithmatic_operator_A, :term) do |lh, operator, rh|
         BLBinaryMethod.new(lh, rh, operator)
```

```
    end
    match(:term) {|a|  a}
end

rule :arithmatic_operator_A do
    match('+') {|_|  'addition'}
    match('-') {|_|  'subtraction'}
end

rule :term do

    match(:term, :arithmetic_operator_B, :mod_expression) do |lh, operator, rh|
        BLBinaryMethod.new(lh, rh, operator)
    end
    match(:mod_expression) {|a|  a}
end

rule :arithmetic_operator_B do
    match('*') {|_| 'multiplication'}
    match('/') {|_| 'divition'}
end

rule :mod_expression do
    match(:mod_expression, :arithmetic_operator_C, :member_call) do |lh, operator, rh|
        BLBinaryMethod.new(lh, rh, operator)
    end
    match(:member_call) {|a|  a}
end


rule :arithmetic_operator_C do
    match('%') {|_|  'modulo'}
end

rule :member_call do
    match(:member_call, '.', :name, '(', :expression_list, ')') {|value, _, func, _, args, _
        BLMemberCall.new(value, func, args)
    }
    match(:member_call, '[', :expression, ']') {|var, _, index, _|
        BLMemberCall.new(var, 'index', [index])
    }
    match(:data) {|a|a}
end

rule :data do
    match('(', :expression, ')') {|_, value, _|  value}
    match(:other_value) {|a|  a}
    match(:func_call) {|a|  a}
    match(:variable_init){|a|a}
    match(:variable) {|a|  a}
    match(:number_value){|a|a}
```

```
end

rule :extra_arg_list do
    match(:extra_arg_list, :data_type) {|a, b| a + [b]}
    match(:data_type){|a|[a]}
end


rule :data_type do
    match(:INTEGER) {|_| BLInteger}
    match(:FLOAT) {|_|  BLFloat}
    match(:BOOL) {|_|  BLBool}
    match(:LIST) {|_| BLList}
end

rule :value do
    match(:number_value){|a|a}
    match(:other_value){|a|a}
end

rule :number_value do
    match(:float_value) {|a|  a}
    match(:int_value) {|a|  a}
end

rule :other_value do
    match(:bool_value) {|a|  a}
    #match(:array_value) {|a|a}
end

rule :int_value do
    match(/\d+/) {|a|  BLConstValue.new(a.to_i, BLInteger)}
    match('-', /\d+/) {|_,a|  BLConstValue.new(-a.to_i, BLInteger)}
end

rule :float_value do
    match(/\d+\.\d+/) {|a|  BLConstValue.new(a.to_f, BLFloat)}
    match('-', /\d+\.\d+/) {|a|  BLConstValue.new(-a.to_f, BLFloat)}
end

rule :bool_value do
    match('true') {|a|  BLConstValue.new(true, BLBool)}
    match('false') {|a|  BLConstValue.new(false, BLBool)}
end

#rule :array_value do
#    match('[', :expression_list, ']') {|_, data, _| data}
#
#end

rule :variable do
```

```
            match(:name) {|name|  BLVarNode.new(name)}
        end

        rule :func_call do
            match(:name, '(', :expression_list, ')') {|name, _, args, _|  BLFuncCall.new(name, args)
            match(:name, '(', ')') {|name, _, _|  BLFuncCall.new(name, []) }
        end

        rule :func_param_list do
            match(:func_param, ',', :func_param_list) {|a, _, b|  a + b}
            match(:func_param) {|a|  a}
        end

        rule :func_param do
            match(:data_type, :name) {|type, name| [BLFuncParam.new(name, type)]}
        end

        rule :expression_list do
            match(:expression_list_segment, ',', :expression_list) {|a, _, b| a + b}
            match(:expression_list_segment) {|a| a}
        end

        rule :expression_list_segment do
            match(:expression) {|a| [a]}
        end
        rule :name do
            match(/[a-zA-Z]\w*/){|a| a}
        end

        rule :begin do
            match('begin')
            match('do')
            match('{')
        end

        rule :end do
            match('end')
            match('}')
        end

        rule :end_line do
            match(';')
        end
    end
end

# For parsing a simple string
def parse_string(string)
    if(@parser.logger.level == Logger::DEBUG)
        puts "=============== Beginning parsing of string ==============="
    end
```

```
        @program = @parser.parse(string)
        if(@parser.logger.level == Logger::DEBUG)
            puts "=============== Parsing of string done ==============="
        end
        if(@print_tree)
            @program.print_tree
        end
    end

    # For parsing a file
    def parse_file(filename)
        if(!filename.end_with?(".bl"))
            raise "File does not have the correct file-ending"
        elsif(!File.exists?(filename))
            raise "Could not find the given file"
        end

        if(@parser.logger.level == Logger::DEBUG)
            puts "=============== Beginning parsing of file ==============="
        end
        file = File.read(filename)
        @program = @parser.parse(file)
        if(@parser.logger.level == Logger::DEBUG)
            puts "=============== Parsing of file done ==============="
        end
        if(@print_tree)
            @program.print_tree
        end
    end

    # Runs the program
    def run()
        puts "==================================  Running program  ==================================
        thing = BLRuntime.new(@program)
        thing.run()
    end

    def log(state = true)
        if state
          @parser.logger.level = Logger::DEBUG
        else
          @parser.logger.level = Logger::WARN
        end
    end
end
```

5.7 rdparse.rb

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

```
# This file is called rdparse.rb because it implements a Recursive
# Descent Parser. Read more about the theory on e.g.
# http://en.wikipedia.org/wiki/Recursive_descent_parser

# 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.
# 2014-02-16 New version that handles { false } blocks and :empty tokens.

require 'logger'

class Rule

  # A rule is created through the rule method of the Parser class, like this:
  #   rule :term do
  #     match(:term, '*', :dice) {|a, _, b| a * b }
  #     match(:term, '/', :dice) {|a, _, b| a / b }
  #     match(:dice)
  #   end

  Match = Struct.new :pattern, :block

  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.nil? ? [] : [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 # pattern.each
      end # matches.each
      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 '#{ma
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
      end
    end

    return match_result
  end
end

class Parser

  attr_accessor :pos
  attr_reader :rules, :string, :logger

  class ParseError < RuntimeError
  end

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

  # Tokenize the string into small pieces
  def tokenize(string)
    @tokens = []
    @string = string.clone
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
      raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |tok|
        match = tok.pattern.match(string)
        # The regular expression of a token has matched the beginning of 'string'
        if match
          @logger.debug("Token #{match[0]} consumed")
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
```

```
        true
      else
        # this token pattern did not match, try the next
        false
      end # if
    end # raise
  end # until
end

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

  # These variables are used to match if the total number of tokens
  # are consumed by the parser
  @pos = 0
  @max_pos = 0
  @expected = []
  # Parse (and evaluate) the tokens received
  result = @start.parse
  # If there are unparsed extra tokens, signal error
  if @pos != @tokens.size
    raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found '#{@tokens[@max_pos]}'"
  end
  return result
end

def next_token
  @pos += 1
  return @tokens[@pos - 1]
end

# Return the next token in the queue
def expect(tok)
  return tok if tok == :empty
  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 # In practise, calls match 1..N times
    @current_rule = nil
  end

  def match(*pattern, &block)
    # Basically calls memberfunction "match(*pattern, &block)
    @current_rule.send(:match, *pattern, &block)
  end

end
```

5.8 run bl.rb

```
require_relative "./parser.rb"

# File for easily running the program and setting some flags for the program that one may want to use
# -tree  : Prints the whole tre structure of the program in a nice tree
# -debug : Enables rdparse debug
# -norun : Makes the program not run after parsing

if(ARGV.length == 0)
    raise "No arguments given"
end

filename = ARGV[ARGV.length - 1]

parser = BaljanLang.new()
parser.log(false)

no_run = false

# Sets all potential flags
ARGV.each_with_index do |arg, index|
    if(index > ARGV.length - 2)
        break
    end
```

```
    if(arg.downcase == "-tree")
        parser.print_tree = true
    elsif(arg.downcase == "-debug")
        parser.log(true)
    elsif(arg.downcase == "-norun")
        no_run = true
    end
end

# Parse the file
parser.parse_file(filename)

# Run the program if no run is off
if(not no_run)
    parser.run()
end
```

5.9 runtime.rb

```
require_relative "./managers.rb"

# Initializing the global vars here because almost everything need them to function
$functions = BLFunctionManager.new()
$memory = BLMemoryManager.new()
$signatures = BLSignatureManager.new()

class BLScope
    attr_reader :signature
    def initialize(signature, parent = nil)
        @parent = parent
        @addresses = {}
        @types = {}
        @signature = signature

        @temp_types = {}
        @refrence_types = {}
    end

    # Add a variable to the scope
    def add(name, type)
        # If it already exists and the variable you tried to add is of the same type as the existing one
        if(@addresses.key?(name))
            addr = @addresses[name]
            if(@types[addr] != type)
                raise RuntimeError.new("Tried initializing already existing variable #{name} with diffe
            else
                return addr
            end
        end
        # If it does not exist then reserve enough memory for the variable
        addr = $memory.reserve(@signature, type.length)
```

```
    @addresses[name] = addr
    @types[addr] = type
    # Return the address of the variable
    return addr
end

# Removesa a variable from the scope
def delete(name)
    if(!@addresses.key?(name))
        raise RuntimeError.new("Tried removing nonexistent variable \"#{name}\"")
    end
    addr = @addresses[name]
    @addresses.delete(name)
    @types.delete(addr)
    type.release(addr)
    return nil
end

# Takes a memory address and returns what variable type is stored there
def address(name)
    if(@addresses.key?(name))
        return @addresses[name]
    end
    if(@parent != nil)
        @parent.address(name)
    end
    raise RuntimeError.new("Tried looking up address for nonexistent variable \"#{name}")
end

# Takes a variable name and returns what address corresponds to that variable
def type(name)
    if(@temp_types.key?(name))
        return @temp_types[name]
    end
    if(@types.key?(name))
        return @types[name]
    end
    if(@parent != nil)
        return @parent.type(name)
    end
    raise RuntimeError.new("Tried looking up type for nonexistent variable with address #{name}")
end

# Creates a temporary variable that is used internally for passing around values
def temp(type, value)
    addr = $memory.reserve(@signature, type.length)
    @temp_types[addr] = type
    type.init(addr, @signature)
    type.set(addr, value)
    return addr
end
```

```
    # Creates a refrence by binding an address to a type, this variable does not have a name
    def refrence(type, addr)
        @types[addr] = type
        return addr
    end

    # Removes all temp values
    def clear_temp()
        @temp_types.each do |addr, type|
            $memory.release(@signature, addr, type.length)
        end
        @temp_types = {}
        if(@parent != nil)
            @parent.clear_temp()
        end
    end

    # Removes one specific temp value
    def remove_temp(addr)
        if(@temp_types.key?(addr))
            $memory.release(@signature, addr, @temp_types[addr].length)
            @temp_types.delete(addr)
        end
        if(@parent != nil)
            @parent.remove_temp(addr)
        end
    end

    # Releases all variables that are connected to this signature
    def cleanup()
        $memory.release(@signature)
        $signatures.release(@signature)
    end

    def to_s
        return "Signature: #{@signature}\nAddresses: #{@addresses}\nTypes: #{@types}\nTemp_types: #{@ter
    end
end

class BLProgram
    def initialize(functions, global_vars)
        @funcs = functions
        @global_vars = global_vars
        @global_scope = nil
    end

    # setup functikon that puts all functions in the function manager and initalizes all global vars wi
    def setup()
        @global_scope = BLScope.new($signatures.get_global())
        @global_vars.each do |var|
```

```
            var.eval(@global_scope)
        end
        $global = @global_scope
        @funcs.each do |func|
            $functions.add(func)
        end
    end

    # Function for printing a tree representation of the program
    def print_tree(level = "")
        puts "Global variables"
        # Printing of the global variables not yet implemented

        puts "Functions"
        @funcs.each_with_index do |func, index|
            temp_level = level
            if(index + 1 < @funcs.length)
                print "#{level} "
                func.print_tree(level + "  ")
            else
                print "#{level} "
                func.print_tree(level + "  ")
            end
        end
    end


end

class BLRuntime
    def initialize(program)
        @program = program
    end

    # Sets upp everything, runs the program, then cleans everything up
    def run()
        # Resetting the global variables in case the have been used and have old data in them
        $functions = BLFunctionManager.new()
        $memory = BLMemoryManager.new()
        $signatures = BLSignatureManager.new()

        @program.setup()

        main = $functions.get(BLFuncSignature.new(:main, nil, []))
        main_scope = BLScope.new($signatures.get(), $global)

        out_value_addr = main.eval(main_scope)
        out_value_type = $global.type(out_value_addr)

        print "Output: "
        out_value_type.output(out_value_addr)
```

```
        puts

        $global.cleanup
    end
end
```