EasyToRead

TDP019 – Projekt: Datorspråk

Maximilian Bragazzi Ihrén
Markus Lewin

Innovativ programmering
Linköpings universitet

Innehållsförteckning

1. Inledning

Detta projekt genomfördes under IP-programmets andra termin, i kursen TDP019 – Projekt: Datorspråk.

Språket vi har implementerat är interpreterat, och tanken bakom det var att göra ett så lättläst språk som möjligt, dvs. att det nästan ska kunna läsas som ren engelska. Med detta har vi riktat in oss på en målgrupp som kan engelska, men som inte nödvändigtvis har erfarenhet inom programmering. Tanken är att denna målgrupp går en kurs (på internet eller i verkligheten), där lärarna går in i de ”djupare” delarna av språket, och kodar eventuella klasser och funktioner som ska användas.

Språkets syntax är influerat av Ruby och Java (eller möjligtvis C++). Att det är influerat av Ruby är inte konstigt, då språket i sig är uppbyggt via ett bibliotek till Ruby vid namn RDParse.

2. Användarhandledning

2.1 Installation

För att installera språket behöver du ha:

- En Ruby-kompilator (språket är testat med Ruby version 1.9.3)
- ”EasyToRead.zip”-filen vilket kan laddas ner via kursens hemsida.

För att installera behöver du endast extrahera .zip-filen till valfri mapp (vi rekommenderar din hemmapp).

Efter att ha skrivit lite kod, kan du då skriva in följande kommando för att exekvera koden:

```
ruby ~/EasyToRead.rb minfil.etr
```

där du kan byta ut ”~”-tecknet mot mappen dit du extraherade filerna (”~” betyder hemmapp).

Om du vill kan du även gå in i din .bashrc-fil (vilken borde ligga i din hemmapp) och klistra in följande:

```
alias etr=”ruby ~/EasyToRead.rb $@”
```

Här kan du också byta ut ”~”-tecknet, likt ovan.

Detta gör att du kan skriva följande kommando för att exekvera koden (efter omstart av terminal):

```
etr minfil.etr
```

2.2 Kodning i EasyToRead

2.2.1 Hello world

Det första man gör när man lär sig ett nytt språk är enligt tradition Hello World-programmet, vilket (helt enkelt) skriver ut ”Hello World” på skärmen. Detta är enkelt att utföra i EasyToRead:

```
          print ”Hello World”;
```

Detta kan klistras in i en ny fil med filändelsen .etr, och kan sedan exekveras. Men det blir nu ett mindre problem! ”print”-funktionen i språket skriver inte ut en ny rad automatiskt! Detta går att lösa på några olika sätt:

```
          print ”Hello World\n”;
```

går att skriva, då alla ”\n”-tecken i en kod omvandlas till nyradstecken vid utskrift. Det finns även en inbyggd funktion för detta:

```
          puts;
```

vilket kan skrivas på raden under. Denna funktion är enkel i det att den endast avslutar en rad. Den kan vara användbar om det du ska skriva ut inte är text, då du inte kan lägga till ”\n”-symbolen i andra typer av variabler.

2.2.2 Semikolon

Något som är viktigt i EasyToRead är användningen av semikolonet (”;”). Denna symbol avslutar ett kommando du vill att programmet ska utföra.

2.2.3 Variabeldefinitioner

Att definiera en variabel är relativt självförklarande i EasyToRead:

```
          define myint as Integer;
```

är en rad kod som skapar en variabel vid namn ”myint” av klassen ”Integer”. Att definiera en variabel följer alltså denna formel:

```
          define <variabelnamn> as <klassnamn>;
```

2.2.4 Funktionsdefinitioner

Att definiera en funktion är likt att definiera en variabel i EasyToRead:

```
          define Hello_World as function {
               print ”Hello World”;
               puts;
          }
```

Den största skillnaden är tillägget av måsvingarna. Det som befinner sig emellan måsvingarna är den kod som kommer exekveras vid funktionens anrop. Funktionen ovan skriver t.ex. ut ”Hello World” på sin egen rad.

Notera att en funktionsdefinition inte avslutas med ett semikolon.

Formeln för funktionsdefinition är alltså:

```
          define <funktionsnamn> as function { <rader av kod> }
```

Som du ser i formeln krävs det inte att varken måsvingarna eller raderna av kod har sin egen rad. Även om detta inte är nödvändigt är det varmt rekommenderat.

2.2.5 Funktionsdefinitioner med parametrar

En funktion i ett kodspråk brukar kunna ta med ett eller flera värden vid sitt anrop. Detta kan göras i EasyToRead på detta sätt:

```
          define multiply as function with parameters(define a as
Integer, define b as Integer) {

               return 'a' * 'b';

          }
```

Vid anrop av denna funktion kan du nu skicka in två heltal, vilka kommer att multipliceras i funktionen och produkten kommer att returneras. Namnen på de värden du skickar in är ”a” och ”b”, vilket kan ses i definitionen. Variablerna skickas in i den ordning de kommer i anropet.

Formeln är:

```
          define <funktionsnamn> as function with parameters(<lista
av variabeldefinitioner>) { <rader av kod> }
```

2.2.6 Klassdefinition

En klassdefinition är lik funktions- och variabeldefinitioner, då man skriver likt följande:

```
          define myclass as class {

               define self as Integer;

               define get_self as function {

                    return 'self';

               }

          }
```

Denna klass innehåller alltså en variabel ”self” av typen Integer, och den har även en funktion ”get_self” vilket returnerar värdet på ”self”-variabeln. Som du kanske ser kan alla funktioner i en klass komma åt alla variabler som finns definierade i den klassen.

Formeln är då:

```
          define <klassnamn> as class {

               <variabel- eller funktionsdefinitioner>

          }
```

2.2.7 Användning av variabler

Variabelhanteringen är unik i EasyToRead. För att hämta ut värdet ur en variabel måste du skriva dess namn inom apostrofer, likt följande:

```
          define a as Integer;

          print 'a';
```

Detta hämtar då ut värdet på variabeln ”a” (försök dock inte exekvera detta kodexempel, då variabeln ej har fått ett värde!).

För att gå in lite djupare kan vi avslöja att när du hämtar värdet ur en variabel med hjälp av apostrofer hämtar kompilatorn värdet på den klassens variabel som heter ”self” (vilket i sin tur får sitt värde hämtat tills man kommer ner till en av grundklasserna). Om ”self”-variabeln inte kan hittas kan inte apostrofer användas för att hämta ut variabelns värde.

2.2.8 Funktionsanrop

Detta är det som mest skiljer EasyToRead från andra språk. Vid ett funktionsanrop är syntaxen på språket flexibelt. T.ex. kan funktionen:

```
          define multiply_$1_by_$2 as function with
parameters(define a as Integer, define b as Integer) {

               return 'a' * 'b';

          }
```

anropas som:

```
          multiply 5 by 7;
```

Detta betyder då alltså att 5:an och 7:an läggs in som inparametrarna till funktionen (i den ordningen). Symbolerna ”$1”, ”$2”, etc. ska alltså ersättas med värdena du vill ha som inparametrar, där ”$1” symboliserar den första inparametern, ”$2” den andra, etc. Det finns ingen begränsning på hur många du kan ha, så länge du har lika många inparametrar.

Det är viktigt att definiera inparametrarna i ordning (du kan inte definiera $2 före $1). Du måste även definiera alla inparametrarna i namnet på funktionen, men om du inte gör det lägger kompilatorn till ”_$<siffra>” för varje parameter du inte skrev med. T.ex. blir namnet på funktionen:

```
          define func as function with parameters(define a as
Integer, define b as Integer) {

               //kod//

          }
```

det här:

```
func_$1_$2
```

Som du kanske även ser kan du byta ut understreck (”_”) i funktionsnamn med mellanrum.

Om du vill skicka in en variabels värde använder du dig av apostrofer som beskrivet i 2.2.8.

2.2.9 Klassfunktionsanrop

För att anropa en funktion från en klass behöver du först en variabel av den klassens typ. Som exempel kommer vi använda oss av Integer:

```
define myint as Integer;
```

I Integer-klassen finns det en funktion definierat likt nedan:

```
define increment_$n as function {

     :self: = 'self' + 1;

}
```

Denna funktion ökar alltså värdet av din Integervariabel med 1. Funktionen anropas på detta vis:

```
increment :a:;
```

Som du nu kanske ser ska du byta ut symbolen ”$n” från funktionsnamnet med variabelnamnet du anropar funktionen ifrån, omringat av kolon (”:”).

Om du vill anropa en klassfunktion från en funktion du definierar i klassen själv anropas den på följande sätt:

```
increment ::;
```

dvs. att du har ett ”tomt” variabelnamn.

Ett klassfunktionsnamn måste innehålla ”$n”-symbolen. Om du inte definierar den själv lägger kompilatorn på ”_from_$n” på namnet. T.ex. blir namnet på funktionen:

```
define func as function {

     //kod//

}
```

följande:

```
func_from_$n
```

Du kan naturligtvis också ha med inparametrar till klassfunktioner. T.ex. finns även denna funktionsdefinition i Integer:

```
          define increment_$n_by_$1 as function with
parameters(define a as Integer) {

               :self: = 'self' + 'a';
}
```

vilket anropas på samma sätt som beskrivet i 2.2.9 och ovan.

2.2.10 If-satser

If-satsens syntax är lik den i andra språk som Java och C++. Exempel:

```
if (5 > 3) {
     print ”5 is larger than 3”;
     puts;
}
```

En if-sats kontrollerar alltså om uttrycket inom parenteserna stämmer (eller om det är ”true” eller ”false”).

Om uttrycket är sant exekveras koden inom måsvingarna.

Om du skulle vilja hantera om uttrycket inte skulle vara sant också kan du använda dig av ”otherwise”:

```
if (5 > 3) {
     print ”5 is larger than 3”;
     puts;
} otherwise {
     print ”Something is very wrong”;
     puts;
}
```

Denna if-sats undersöker alltså om 5 är större än 3. Om det stämmer skriver den ut ”5 is larger than 3”, men om det inte stämmer skriver den ut ”Something is very wrong”.

Om du vill testa något annat om din första if-sats inte stämmer kan du då skriva in en till if-sats mellan måsvingarna för ”otherwise”:

```
if (5 > 3) {
     //kod//
} otherwise {
     if (3 > 5) {
          //kod//
     }
}
```

Detta kan upprepas så många gånger du behöver.

2.2.11 While-satser

While-satser, likt if-satser, är likt i syntax till t.ex. Java och C++:

```
while (1 == 1) {
     print ”1 equals 1”;
     puts;
}
```

En while-sats exekverar koden mellan måsvingarna så länge uttrycket mellan parenteserna stämmer.

Försök inte exekvera exempelkoden ovan dock, då detta är en oändlig loop (eftersom uttrycket alltid kommer att stämma)!

2.2.12 Return-satser

En funktion kan ha en ”return”-sats i sig, vilket eventuellt skickar tillbaka ett värde dit där funktionen anropades. En return-sats kan även vara tom, dvs. inte returnera något.

När programmet kommer fram till att exekvera en return-sats kommer inte några rader kod efter satsen i funktionen att exekveras, då funktionsanropet avslutas.

En funktion kan bara returnera en sorts returvärde, vilket sätts till vad det första returvärdet i funktionsanropet är. Dvs, funktionen:

```
define myfunc as function {
     if (5 > 6) {
          return ”Error”;
     } otherwise {
          return 5;
     }
}
```

kan inte användas, då den endast kan returnera saker av typen String (texten mellan citattecken), men den kommer att försöka returnera 5, vilket är av typen Integer.

Du kan även returnera variabler, genom att hämta värdet ur den som i 2.2.8.

2.2.13 Inbyggda klasser

Dessa klasser finns färdigbyggda i språket:

- String

  Detta är text. Det definieras med citattecken runt text.

- Integer

  Detta är heltal. Definieras genom att skriva ett heltal.

- Float

  Detta är decimaltal. Definieras genom att skriva ett decimaltal med separatorn ”.” mellan heltal och decimaler. T.ex.: 5.3

- Boolean

  Denna klass kan endast innehålla värden ”true” eller ”false”. Kan ses som en ”knapp” som är i ”på”- eller ”av”-läge

- Array

  Denna klass är en lista som kan innehålla var och en av de ovanstående klasserna. Definieras genom att skriva hakparenteser runt hela listan, och separera olika listelement med kommatecken. T.ex.: [5, ”Hej”, false].

2.2.14 Kommentarer

Kommentarer i kod är saker som hjälper programmeraren att dokumentera sin kod bättre. Text eller kod innanför kommentarer exekveras aldrig. Det kan därför användas för att temporärt ta bort kod ur ditt projekt. Ett exempel på en kommentar i EasyToRead är:

```
define a as Integer;

set :a: to 5; //Sets a to 5//

//set :a: to 10;

set :a: to 20;//
```

Texten innanför ”//” är då kommentarer, vilket inte kommer exekveras. Det koden ovan kommer att göra är att skapa en variabel ”a”, och sätta dess värde till 5.

3. Systemdokumentation

3.1 Översikt

Vårt system är byggt utifrån att ha två olika ”parsers”, en som läser koden och en som exekverar den. Den första parsern läser in hela användarens kod som en enda lång sträng, vilket parsrar uttryck för uttryck och sparar undan alla klass-, funktions- och variabeldeklarationer i speciella variabler, och översätter dessa till en mycket mer strikt syntax som den andra parsern kan läsa. Den första parsern består alltså av en lexer och en parser/översättare, medans den andra består av en lexer och en evaluerande/exekverande del.

3.2 BNF-grammatik

3.2.1 BNF-grammatik för readern

```
<program> ::= <contents>
<contents> ::= <content> <contents>
                  | <content>
<content> ::= <classdefs>
                  | <funcdefs>
                  | <stmts>
<classdefs> ::= <classdefs> <classdef>
                  | <classdef>
<classdef> ::= 'define' <NAME> 'as' 'class' <inheritance> '{' '}'
                  | 'define' <NAME> 'as' 'class' '{' '}'
                  | 'define' <NAME> 'as' 'class' <inheritance> '{' <classcont> '}'
                  | 'define' <NAME> 'as' 'class' '{' <classcont> '}'
<classcont> ::= <memfuncdefs> <classcont>
                  | <memvardefs> <classcont>
                  | <memfuncdefs>
                  | <memvardefs>
<memfuncdefs> ::= <memfuncdefs> <memfuncdef>
                        | <memfuncdef>
<memfuncdef> ::= <funcdef>
<memvardefs> ::= <memvardefs> <memvardef>
                       | <memvardef>
<memvardef> ::= <vardef> <SEP>

<funcdefs> ::= <funcdefs> <funcdef>
                  | <funcdef>
<funcdef> ::= 'define' <NAME> 'as' 'function' 'with' 'parameters' '(' <parameters> ')' <codeblock>
```

```
                    | 'define' <NAME> 'as' 'function' <codeblock>
<parameters> ::= <vardef> ',' <parameters>
                    | <vardef>
<codeblock> ::= '{' <stmts> '}'
                     | '{' '}'
<stmts> ::= <stmts> <stmt>
                 | <stmt>
<stmt> ::= <vardef> <SEP>
               | <ifstmt>
               | <whilestmt>
               | <setstmt>
               | <equalitystmt> <SEP>
               | < mathstmt> <SEP>
               | <logicstmt> <SEP>
               | <printstmt>
               | <returnstmt>
               | <func_call>
<mathstmt> ::= <mathstmt> '+' <mathstmt>
                    | <mathstmt> '-' <mathstmt>
                    | <mathstmt> '*' <mathstmt>
                    | <mathstmt> '/' <mathstmt>
                    | <mathstmt> '%' <mathstmt>
                    | <mathstmt> '**' <mathstmt>
                    | '(' <mathstmt> ')'
                    | /\d+/
                    | /\d+\.\d+/
                    | /'\w+'/
<logicstmt> ::= <logicstmt> '&&' <logicstmt>
                    | <logicstmt> '||' <logicstmt>
                    | '!' <logicstmt>
               | '(' <logicstmt> ')'
               | 'true'
               | 'false'
               | /'\w+'/
<equalitystmt> ::= <equalitystmt> '<' <equalitystmt>
                   | <equalitystmt> '>' <equalitystmt>
                   | <equalitystmt> '==' <equalitystmt>
                   | '(' <equalitystmt> ')'
                   | <DEFAULT_CLASS>
                   | /'\w+'/
                   | <mathstmt>
                   | <logicstmt>
<DEFAULT_CLASS> ::= /\[[[^\]],]*[^\]]*\]/
                          | /"[^"]*"/
                          | <mathstmt>
                          | <logicstmt>
<setstmt> ::= /:\w+:/ '=' <DEFAULT_CLASS> <SEP>
              | /:\w+:/ '=' <func_call>
<printstmt> ::= 'print' <DEFAULT_CLASS> <SEP>
               | 'print' <equalitystmt> <SEP>
               | 'print' /'\w+'/ <SEP>
               | 'print' <func_call>
<returnstmt> ::= 'return' <DEFAULT_CLASS> <SEP>
               | 'return' <equalitystmt> <SEP>
               | 'return' /'\w+'/ <SEP>
               | 'return' <func_call>

<ifstmt> ::= 'if' '(' <expr> ')' <codeblock> 'otherwise' <codeblock>
             | 'if' '(' <expr> ')' <codeblock>
<whilestmt> ::= 'while' '(' <expr> ')' <codeblock>
```

```
<expr> ::= <equalitystmt>
              | /[^)]+/ <expr>
              | /[^)]+/
<vardef> ::= 'define' <NAME> 'as' <NAME>
<inheritance> ::= '<' <cnamelist>
<cnamelist> ::= <cnamelist> ',' <NAME>
                   | <NAME>
<func_call> ::= /[^;{}]+/ <func_call>
                   | /[^;{}]+/ <SEP>
<NAME> ::= /[^\s({}),;"]+/
<PATH> ::= /\".+\.etr\"/
<SEP> ::= ';'
```

3.2.2 BNF-grammatik för parsern

```
<program> ::= <stmts>
<stmts> ::= <stmts> <stmt>
               | <stmt>
<stmt> ::= <ifstmt>
              | <whilestmt>
              | <setstmt>
              | <equalitystmt>
              | <logicstmt>
              | <mathstmt>
              | <printstmt>
              | <returnstmt>
              | <arrayaddstmt>
              | <func_call>
<arraygetstmt> ::= /:\w+:\[\d+\]/
                        | /:\w+:\['\w+'\]/
<arrayaddstmt> ::= /:\w+:<</ <DEFAULT_CLASS>
```

```
                | /:\w+:<</ <GETVAR>
                | /:\w+:<</ <arraygetstmt>
<ifstmt> ::= 'if' <DIGIT>
<whilestmt> ::= 'while' <DIGIT>
<setstmt> ::= <USEVAR> '=' <arraygetstmt>
            | <USEVAR> '=' /\[[[^\]],]*[^\]]*\]/
            | <USEVAR> '=' /"[^"]*"/
            | <USEVAR> '=' <mathstmt>
            | <USEVAR> '=' <logicstmt>
            | <USEVAR> '=' <GETVAR>
            | <USEVAR> '=' <func_call>
<mathstmt> ::= <plusstmt>
<plusstmt> ::= <plusstmt> '+' <minusstmt>
             | <minusstmt>
<minusstmt> ::= <minusstmt> '-' <multstmt>
              | <multstmt>
<multstmt> ::= <multstmt> '*' <divstmt>
             | <divstmt>
<divstmt> ::= <divstmt> '/' <modstmt>
            | <modstmt>
<modstmt> ::= <modstmt> '%' <powstmt>
             | <powstmt>
<powstmt> ::= <powstmt> '**' <mathexpr>
             | <mathexpr>
<mathexpr> ::= '(' <mathstmt> ')'
             | <GETVAR>
             | <DIGIT>
<logicstmt> ::= <orstmt>
<orstmt> ::= <orstmt> '||' <andstmt>
           | <andstmt>
```

```
<andstmt> ::= <andstmt> '&&' <notstmt>
                | <notstmt>
<notstmt> ::= '!' <notstmt>
                | <logicexpr>
<logicexpr> ::= '(' <logicstmt> ')'
                | 'true'
                | 'false'
                | <GETVAR>
<equalitystmt> ::= <equalitystmt> '<' <equalitystmt>
                    | <equalitystmt> '>' <equalitystmt>
                    | <equalitystmt> '==' <equalitystmt>
                    | '(' <equalitystmt> ')'
                    | <DEFAULT_CLASS>
                    | <GETVAR>
                    | <mathexpr>
                    | <logicexpr>
<printstmt> ::= 'print' <arraygetstmt>
                | 'print' <DEFAULT_CLASS>
                | 'print' <GETVAR>
                | 'print' <func_call>
<returnstmt> ::= 'return' <DEFAULT_CLASS>
                  | 'return' <equalitystmt>
                  | 'return' <arraygetstmt>
                  | 'return' <func_call>
<func_call> ::= /[^;{}]+/
<DEFAULT_CLASS> ::= /\[[[^\]],]*[^\]]*\]/
                        | /"[^"]*"/
                        | <mathstmt>
                        | <logicstmt>
<GETVAR> ::= /'\w+'/
```

<USEVAR> ::= /:\w+:/

<DIGIT> ::= /\d+\.\d+/

| /\d+/

3.3 Kodstandard

I parser-koden följer vi bara den generella standarden för Ruby-kod, dvs. indentering vid kodblock, funktioner har inga stora bokstäver, funktionsnamn har understreck i namnet, klasser börjar med stor bokstav, etc.

I ETR använder vi en standard som liknar den i C++ när det gäller indentering och namngivning till saker. Dock har ju funktionsnamn sin egen, (relativt) unika syntax, vilket finns beskrivet i kapitel 2. Det finns dock inget krav på någon standard i ETR, men det är rekommenderat att använda den standard vi har satt för enklare läsning av koden (vilket var grundidén).

4. Reflektion

Från början var vår tanke att vårt språk skulle vara relativt likt det vi kom fram till i slutändan. Då hade vi dock ingen aning om hur svårt det skulle bli att ha funktionsanrop utan att indikera parametrar och variabler du anropar ifrån. Från början såg alltså t.ex. ”set :x: to 5;” ut som ”set x to 5.”. Vi ändrade oss också från att ha punkter som separator, då vi kom på att det skulle kunna läsas som ett flyttal av parsern.

Från detta såg vi att man inte kan förutse alla svårigheter som ett projekt kan innebära i planeringsstadiet, och speciellt inte på en sådan här projektskala. Därför måste man vara beredd på att göra uppoffringar kring sitt projekt.

Efter att ha klurat ut hur man ska fixa detta problem (med ”lite” hjälp) gick det snabbt framåt. Vi blev klara med den första parsern efter en vecka eller två, men var inte säkra på hur vi skulle fortsätta från det. Det enda vi hade gjort för att exekvera kod i RDParser:n tidigare var i TDP007, där man gjorde det direkt i BNF-koden, vi kom fram med att vi skulle göra en ny parser som exekverade det kodträd som den första parsern returnerade.

Efter att ha hört hur andra hade gjort via noder och liknande verkade vår version inte lika effektiv, men vid det tillfället var det för sent att ändra sig, vi körde på. Vi känner nu att vi kanske borde ha kollat hur tidigare projekt hade gjort (eller frågat handledaren) om hur vi skulle ha gått tillväga efter den första parsern.

Det enda vi inte lade till från den ursprungliga planen var for-loopar och en annan loop som bara loopade koden ett visst antal gånger (utan att behöva skapa en variabel som i for-loopar).

Dessa blev inte implementerade pga. tidsbrist, eftersom att implementera klasser och dess scopes tog mycket längre tid än vad vi hade förutsett. Vi antog att det bara skulle bli en sorts förlängning på funktioner, men det blev en nästan helt annorlunda princip.

Dock är vi väldigt nöjda med resultatet! Det funkar som det ska (om lite långsamt) och är tillräckligt likt vårat mål vi hade i början att vi känner att det uppfyller våra förhoppningar.

5. Kod

5.1 EasyToRead.rb

```ruby
# -*- coding: utf-8 -*-
require './parser.rb'
require './nonlanguagethings.rb'

class ETR

  def initialize
    @@classes = {}
    @@vars = {}
    @@funcs = {}
    @@maincontent = []
    #@@includes = []
    @@includesdone = []
    @@ifstmts = []
    @@whilestmts = []
    @@defaultclasses = {}
    @@defaultfuncs = {}
    @@returntype = "Integer"

    @@returnstmt = false

    # Match everything between [ and ] that is not ],
    # with a possibility of something like "a,"
    # and "a"
    @@arrayRegex = /\[[[^\]],]*[^\]]*\]/


####################################################################
############
#
# READER STARTS HERE
#
####################################################################
############
    @reader = Parser.new("EasyToRead") do
      everythingTokenRegex = /[^\s({}),;"]+/

      # Comments, throw. Match everything between // and // that is
not //
      token(/\/\/[^\/\/]+\/\//)

      # Special symbols
      token(/(\(|\))/) {|a| a } # Match ( and )
      token(/({|})/) {|a| a } # Match { and }
      token(/,/) {|a| a }
      token(/;/) {|a| a }
      token(/"[^"]*"/) {|a| a } # Match everything between " and "
that is not "
      token(/\[[[^\]],]*[^\]]*\]/) {|a| a }# Match all arrays
      token(/\s+/)

      token(everythingTokenRegex) {|a| a } # Everything else

      start :program do
        match(:contents)
      end

      rule :contents do
        match(:contents, :content) {|a, b| [a, b]}
        match(:content)
      end

      def check_if_var_exists(var)
        # Checks if the parameter is not a constant (therefore a
variable),
        # and in that case checks if it has been initiated, thus
preventing
        # variable use before definition.
        if (!@@vars.has_key?(var) && !var.is_boolean? && !
var.is_digit? && !var.is_string?)
          if (var.is_array?)
            var = var[1..-2]
            var.split(",").each do |elem|
              check_if_var_exists(elem.strip)
            end
          else
            raise "Variable \"#{var}\" used before initiation"
          end
        end
      end

      rule :content do
        # Content is everything the program can contain
=begin
        match(:includes) {|a|
          # Saves all the names of the files the user has included
to an array
          @@includes = [@@includes, a].flatten!
          a }
=end
        match(:classdefs)
        match(:funcdefs) {|a|
          # Saves all function definitions to the @@funcs variable.
          # If the match matches more than one function definition,
          # it loops through the resulting array
          if (a.class == Array)
            a.flatten!
            a.each do |entry|
              @@funcs[entry.general_name] = entry
            end
          else
            @@funcs[a.general_name] = a
          end
          a }
        match(:stmts) {|a|
          # Statements are either variable definitions,
          # of "main content" (function calls, mathematical
expressions,
          # print statements, etc). Also prevents that a variable
is declared
          # more than once (in a scope).
          # Variables are saved in the Hash "@@vars", and
          # "main content" in an Array by the same name
          if (a.class == Array)
            a.flatten!
            a.each do |entry|
              if (entry.class == VarHolder)
                if (@@vars.has_key?(entry.name))
                  raise "The variable '#{entry.name}' is declared
more than once"
                else
                  @@vars[entry.name] = entry
                end
              else
                get_vars_from_func_call(entry).each do |var|
                  check_if_var_exists(var.strip)
                end
                @@maincontent << entry
              end
            end
          elsif (a.class == VarHolder)
            raise "The variable '#{a.name}' is declared more than
once" if (@@vars.has_key?(a.name))
            @@vars[a.name] = a
          else
            get_vars_from_func_call(a).each do |var|
              check_if_var_exists(var.strip)
            end

            @@maincontent << a
          end
          a }
      end

      rule :classdefs do
        # Catches all class definitions
        match(:classdefs, :classdef) {|a, b| [a, b]}
        match(:classdef)
      end

      def classdeffunc(name, inherit, cont)
```

```
        # Since we needed so many different class definitions,
        # we decided to make the class definition into a function.
        ch = ClassHolder.new
        # Function checks if the class has been defined already,
        # and updates that class in that case.
        if (@@classes.has_key?(name))
          ch = @@classes[name]
        elsif (@@defaultclasses.has_key?(name))
          ch = @@defaultclasses[name]
        end
        ch.name = name
        ch.inherits << inherit
        ch.inherits.flatten!

        # Goes through all the inherits and adds all the functions
to the
        # current class' member function Hash, with the key being
the functions
        # general name, and the value being the name of the class
it can be
        # found in. This is to prevent a lot of looping when a call
is made.
        ch.inherits.each do |inherit_name|
          if (!@@classes.has_key?(inherit_name))
            raise "The inherit #{inherit_name} is being inherited
before the class is defined!"
          end

          @@classes[inherit_name].memfuncs.each do |key, val|
            ch.memfuncs[key] = inherit_name
          end
        end

        # Loops through all the content in the class, and adds all
the variables
        # to the variable "memvars", and all the
        # functions to the variable "memfuncs".
        cont.flatten!
        cont.each do |entry|
          if (entry.class == VarHolder)
            if (ch.memvars.has_key?(entry.name))
              raise "The variable '#{entry.name}' is declared more
than once in the class '#{name}'"
            else
              ch.memvars[entry.name] = entry
            end
          elsif (entry.class == FuncHolder)
            ch.memfuncs[entry.general_name] = entry
          end
        end

        # Adds the variable "__class__" to every class,
        # containing the class' name, and the function
        # to get the variable.
        ch.memvars["__class__"] = VarHolder.new("__class__",
"String", name)
        func = FuncHolder.new("$n.class", [], ["return
'__class__'"])
        ch.memfuncs[func.general_name] = func

        # Sets the return type for all the member functions
        ch.memfuncs.each do |name, func|
          # Skip if the function is in another class
          next if (func.class == String)

          # Gets the name (or the constant) of the first return
value,
          # removes any "trash" symbols and then skips if the name
is empty.
          return_var = func.get_first_return_var(@@ifstmts,
@@whilestmts)
          return_var = return_var[1..-2] if (return_var[0] =~ /(')/
&& return_var[-1] =~ /(')/)
          next if (return_var.empty?)

          # Sets "returntype" if the variable is one
          # of the member variables, or a constant
          func.returntype = return_var.get_return_type(ch.memvars)

          #if (ch.memvars.has_key?(return_var))
          #  func.returntype = ch.memvars[return_var].type
          #else
          # If "returntype" still is empty, check if the
          # return value is defined in the function.
          if (func.returntype == nil || func.returntype.empty?)
            func.parameters.each do |param|
              func.returntype = param.type if (param.name ==
return_var)
            end
            if (func.returntype == "")
              func.contents.each do |elem|
                func.returntype = elem.type if (elem.class ==
VarHolder && elem.name == return_var)
              end
            end
          end
        end

        @@classes[ch.name] = ch
      end

      rule :classdef do
        # Matches classes with inheritance but no content,
        # no inheritance or content,
        # inheritance and content,
        # or no inheritance and content.
        match('define', :NAME, 'as', 'class', :inheritance, '{',
'}') {|_, name, _, _, inherit, _, cont, _|
          classdeffunc(name, inherit, [])
          [name, inherit, cont] }

        match('define', :NAME, 'as', 'class', '{', '}') {|_, name,
_, _, _, cont, _|
          classdeffunc(name, [], [])
          [name, cont] }
        match('define', :NAME, 'as', 'class', :inheritance, '{',
:classcont, '}') {|_, name, _, _, inherit, _, cont, _|
          classdeffunc(name, inherit, cont)
          [name, inherit, cont] }

        match('define', :NAME, 'as', 'class', '{', :classcont, '}')
{|_, name, _, _, _, cont, _|
          classdeffunc(name, [], cont)
          [name, cont] }
      end

      rule :classcont do
        # A class can contain member function definitions
        # and member variable definitions
        match(:memfuncdefs, :classcont) {|a, b| [a, b]}
        match(:memvardefs, :classcont) {|a, b| [a, b]}
        match(:memfuncdefs)
        match(:memvardefs)
      end

      rule :memfuncdefs do
        # Captures all the member functions
        match(:memfuncdefs, :memfuncdef) {|a, b|
          if (a.class == Array)
            a = (a << b).flatten!
            a
          else
            [a, b]
          end }
        match(:memfuncdef)
      end

      rule :memfuncdef do
        # A member function name has to include
        # the symbol "$n", so if it doesn't, it
        # gets added.
        match(:funcdef) {|a|
          if (!a.name.include?("$n"))
            a.name += "_from_$n"
          end

          [a] }
      end

      rule :memvardefs do
        # Captures all member variables
        match(:memvardefs, :memvardef) {|a, b| [a, b].flatten! }
        match(:memvardef)
      end
      rule :memvardef do
        # Captures all member variable defintions one by one
        match(:vardef, :SEP) {|a, _| [a] }
      end

      rule :funcdefs do
        # Captures all functions (outside of classes)
        match(:funcdefs, :funcdef) {|a, b| [a, b]}
        match(:funcdef)
      end

      rule :funcdef do
        # Matches function definitions with and without parameters.
        match('define', :NAME, 'as', 'function', 'with',
'parameters', '(', :parameters, ')', :codeblock) {|_, name, _, _,
_, _, _, param, _, cont|
          fh = FuncHolder.new
          fh.parameters = param
          fh.contents = cont

          # Since all parameters have to be in the function name
          # if they aren't there, it adds them.
          (1..param.length).each do |i|
            if (!name.include?("$#{i}"))
              name += "_$#{i}"
            end
          end
          fh.name = name

          # Tries to set the return type
          return_var = fh.get_first_return_var(@@ifstmts,
@@whilestmts)
          fh.returntype = return_var.get_return_type(@@vars)

          # If still not set, checks parameters and content
          # for the returned variable definition.
          if (fh.returntype == nil)
            param.each do |parameter|
              fh.returntype = parameter.type if (parameter.name ==
return_var[1..-2])
            end

            if (fh.returntype == nil)
              cont.each do |elem|
                fh.returntype = elem.type if (elem.class ==
VarHolder && elem.name == return_var[1..-2])
              end
            end
          end

          fh }
        match('define', :NAME, 'as', 'function', :codeblock) {|_,
name, _, _, cont|
          fh = FuncHolder.new
          fh.name = name
          fh.contents = cont

          # Tries to set the return type
          return_var = fh.get_first_return_var(@@ifstmts,
@@whilestmts)
          fh.returntype = return_var.get_return_type(@@vars)

          # If still not set, checks the content
          # for the returned variable definition
          if (fh.returntype == nil)
            cont.each do |elem|
              fh.returntype = elem.type if (elem.class == VarHolder
&& elem.name == return_var[1..-2])
            end
          end

          fh }
      end

      rule :parameters do
        # For example: define a as Integer, define b as Boolean
        match(:vardef, ',', :parameters) {|a, _, b| [a, b].flatten!
}
        match(:vardef) {|a| [a] }
      end
      rule :codeblock do
        # Matches codeblocks with and without contents
        match('{', :stmts, '}') {|_, a, _|
          if (a.class == Array)
            a
          else
            [a]
          end }

        match('{', '}') {|_, _| []}
      end

      rule :stmts do
        # Captures all statements
        match(:stmts, :stmt) {|a, b|
          if (a.class == Array)
            a << b
            a
          else
            [a, b]
          end }
        match(:stmt)
      end

      rule :stmt do
        match(:vardef, :SEP) {|a, _| a } # Variable definitions
        match(:ifstmt) # If statements
        match(:whilestmt) # While statements
        match(:setstmt) # Set statements (a = 5)
        match(:equalitystmt, :SEP) {|a, _| a } # Equality
statements (a == b, a < b)
        match(:mathstmt, :SEP) {|a, _| a } # Math statements
        match(:logicstmt, :SEP) {|a, _| a } # Logic statements
        match(:printstmt) {|a, _| a } # Print statments
        match(:returnstmt) {|a, _| a } # Return statements
        match(:func_call) # Function calls, matched last since it
matches everything.
      end

      rule :mathstmt do
        # Matches all math statements, digits and variables
        match(:mathstmt, '+', :mathstmt) {|a, _, b| "#{a} + #{b}" }
        match(:mathstmt, '-', :mathstmt) {|a, _, b| "#{a} - #{b}" }
        match(:mathstmt, '*', :mathstmt) {|a, _, b| "#{a} * #{b}" }
        match(:mathstmt, '/', :mathstmt) {|a, _, b| "#{a} / #{b}" }
        match(:mathstmt, '%', :mathstmt) {|a, _, b| "#{a} % #{b}" }
        match(:mathstmt, '**', :mathstmt) {|a, _, b| "#{a} ** #{b}"
}
        match('(', :mathstmt, ')') {|_, a, _| "( #{a} )" }
        match(/\d+/)
        match(/\d+\.\d+/)
        match(/'\w+'/)
      end

      rule :logicstmt do
        # Matches all logic statements, true, false and variables
        match(:logicstmt, '&&', :logicstmt) {|a, _, b| "#{a} &&
#{b}" }
        match(:logicstmt, '||', :logicstmt) {|a, _, b| "#{a} ||
#{b}" }
        match('!', :logicstmt) {|_, a| "! #{a}" }
        match('(', :logicstmt, ')') {|_, a, _| "( #{a} )" }
        match('true')
        match('false')
        match(/'\w+'/)
      end

      rule :equalitystmt do
        # Matches all equality statements, variables,
        # default classes, math expressions and logic expressions
        match(:equalitystmt, '<', :equalitystmt) {|a, _, b| "#{a} <
#{b}" }
        match(:equalitystmt, '>', :equalitystmt) {|a, _, b| "#{a} >
#{b}" }
        match(:equalitystmt, '==', :equalitystmt) {|a, _, b| "#{a}
== #{b}" }
        match('(', :equalitystmt, ')') {|_, a, _| "( #{a} )" }
        match(:DEFAULT_CLASS)
        match(/'\w+'/)
        match(:mathstmt)
        match(:logicstmt)
      end
```

```
rule :DEFAULT_CLASS do
  # Matches all default class constants
  match(@@arrayRegex) {|a|
    if (a.is_array?)
      a
    else
      nil
    end }
  match(/"[^"]*"/) {|a|
    if (a.is_string?)
      a
    else
      nil
    end }
  match(:mathstmt)
  match(:logicstmt)
end

rule :setstmt do
  # Matches all set statements for constants,
  # variables (through DEFAULT_CLASS) and function calls
  match(/:\w+:/, '=', :DEFAULT_CLASS, :SEP) {|a, _, b, _|
    if (b.is_array?)
      "#{a} = #{b.gsub(" ", "")}"
    else
      "#{a} = #{b}"
    end }
  match(/:\w+:/, '=', :func_call) {|a, _, b| "#{a} = #{b}" }
end

rule :printstmt do
  # Matches all print statements for default class constants,
  # variables, equality statements and function calls
  match('print', :DEFAULT_CLASS, :SEP) {|_, a|
    if (a.is_array?)
      "print #{a.gsub(" ", "")}"
    else
      "print #{a}"
    end }
  match('print', :equalitystmt, :SEP) {|_, a, _| "print #{a}"
}
        match('print', /'\w+'/, :SEP) {|_, a| "print #{a}" }
        match('print', :func_call) {|_, a| "print #{a}" }
      end

      rule :returnstmt do
        # Matches all return statements for default class
constants,
        # variables, equality statements and function calls
        match('return', :DEFAULT_CLASS, :SEP) {|_, a|
          if (a.is_array?)
            "return #{a.gsub(" ", "")}"
          else
            "return #{a}"
          end }
        match('return', :equalitystmt, :SEP) {|_, a, _| "return
#{a}" }
        match('return', /'\w+'/, :SEP) {|_, a| "return #{a}" }
        match('return', :func_call) {|_, a| "return #{a}" }
      end

      rule :ifstmt do
        # Matches simple if statements and if-otherwise statements
        match('if', '(', :expr, ')', :codeblock, 'otherwise',
:codeblock) {|_, _, expr, _, cont, _, othercont|
          ih = IfHolder.new(cont, expr, othercont)

          # Saves the index the if statement will get
          index = @@ifstmts.length

          # Loops through all the previously defined if statements,
          # and checks if one equals another, and saves it as that
          # if statement instead of a new entry.
          @@ifstmts.length.times do |i|
            if (ih.equals?(@@ifstmts[i]))
              index = i
              break
            end
          end

          # Adds the if statement to the array, and returns the if
identifier
          @@ifstmts << ih if (index == @@ifstmts.length)
          "if #{index}" }

        match('if', '(', :expr, ')', :codeblock) {|_, _, expr, _,
cont|
          ih = IfHolder.new(cont, expr)

          # Saves the index the if statement will get
          index = @@ifstmts.length

          # Loops through all the previously defined if statements,
          # and checks if one equals another, and saves it as that
          # if statement instead of a new entry.
          @@ifstmts.length.times do |i|
            if (ih.equals?(@@ifstmts[i]))
              index = i
              break
            end
          end

          # Adds the if statement to the array, and returns the if
identifier
          @@ifstmts << ih if (index == @@ifstmts.length)
          "if #{index}" }
      end

      rule :whilestmt do
        match('while', '(', :expr, ')', :codeblock) {|_, _, expr,
_, cont|
          wh = WhileHolder.new(cont, expr)

          # Saves the index the while statement will get
          index = @@whilestmts.length

          # Loops through all the previously defined while
statements,
          # and checks if one equals another, and saves it as that
          # while statement instead of a new entry.
          @@whilestmts.length.times do |i|
            if (wh.equals?(@@whilestmts[i]))
              index = i
              break
            end
          end

          # Adds the while statement to the array,
          # and returns the while identifier
          @@whilestmts << wh if (index == @@whilestmts.length)
          "while #{index}" }
      end

      rule :expr do
        # An expression is handled like a function call
        match(:equalitystmt)
        match(/[^)]+/, :expr) {|a, b|
          a += "_" + b
          a }
        match(/[^)]+/)
      end

      rule :vardef do
        # For example: define a as Integer
        match('define', :NAME, 'as', :NAME) {|_, name, _, cname|
          vh = VarHolder.new
          vh.name = name
          vh.type = cname

          # Check if the class you are defining your variable as
          # is defined, and then sets the variables value
          # to a Hash of the member variables of that class
          if (!@@defaultclasses.empty? && cname != "function" &&
cname != "class")
            if (@@classes.has_key?(cname))
              vh.value = {}
              @@classes[cname].memvars.each do |key, val|
                vh.value[key] = val.dup unless (key == "__class__")
              end
            elsif (@@defaultclasses.has_key?(cname))
              vh.value = {}
              @@defaultclasses[cname].memvars.each do |key, val|
                vh.value[key] = val.dup unless (key == "__class__")
            end
          else
            raise "The classname #{cname} is used before
definition or not defined at all!"
          end
        end
        vh }
    end

    rule :inheritance do
      # Matches all inheritances
      match('<', :cnamelist) {|_, a|
        if (a.class == Array)
          a
        else
          [a]
        end}
    end

    rule :cnamelist do
      # Matches all NAMEs in a list (separated by ",")
      match(:cnamelist, ',', :NAME) {|a, _, b|
        if (a.class == Array)
          a << b
          a
        else
          [a, b]
        end }
      match(:NAME)
    end

    rule :func_call do
      # A function call matches anything up until a SEP.
      # Combines words with "_" and
      # removes all whitespaces.
      match(/[^;{}]+/, :func_call) {|a, b|
        a += "_" + b
        a.gsub(" ", "\\s") }
      match(/[^;{}]+/, :SEP) {|a, _| a }
    end
```

```
    rule :NAME do
      # Matches everything you would imagine you
      # can call a variable/function/class
      match(everythingTokenRegex)
    end

    rule :PATH do
      # For example: "myfile.etr"
      match(/\".+\.etr\"/)
    end

    rule :SEP do
      # This is its own rule for easy changing of the separator.
      match(';')
    end

  end

####################################################################
############
#
# PARSER STARTS HERE
#
####################################################################
############
  @@parser = Parser.new("EasyToRead") do

    token(/;/) # Statement separator. Throw
    token(/\s+/) # Spaces. Throw
    token(/_/) # Extra underlines. Throw
    token(@@arrayRegex) {|m| m } # Array
    token(/"[^"]*"/) {|m| m } # Strings
    token(/\d+\.\d+/) {|m| m } # Floats
    token(/\d+/) {|m| m } # Integers
    token(/[()]/) {|m| m } # Parenthesis
    token(/(\*\*|\+|-|\/|\*|%)/) {|m| m } # Math signs
    token(/(true|false)/) {|m| m } # true / false
    token(/(==|<|>|=|!|&&|\|\|)/) {|m| m } # logic statements
    token(/(print|return|if|otherwise|while)/) {|m| m } # Builtin
words
    token(/:\w+:<</) {|m| m } # Array append symbol
    token(/[^;\s]+/) {|m| m } # Everything else

    def set_array_content(array)
      # Sets the content of an array to the correct class
      # Numbers are set to Integers,
      # True/False are set to Booleans, etc.
      #
      # When you have an array in an array,
      # the function turns recursive.
      array.length.times do |i|
        next if (array[i].class != String)
        array[i] = array[i].strip
        if (array[i][0] == "'" && array[i][-1] == "'")
          array[i] = @@vars[array[i][1..-
2]].get_lowest_self.value
        elsif (array[i].is_int?)
          array[i] = array[i].to_i
        elsif (array[i].is_float?)
          array[i] = array[i].to_f
        elsif (array[i].is_boolean?)
          array[i] = array[i].to_b
        elsif (array[i].is_string?)
          array[i] = array[i][1..-2]
        elsif (array[i].is_array?)
          array[i] = set_array_content(array[i])
        end
      end

      return array
    end

    start :program do
      match(:stmts)
    end

    rule :stmts do
      match(:stmts, :stmt) {|a, b| [a, b] }
      match(:stmt)
    end
```

```
    rule :stmt do
      match(:ifstmt)
      match(:whilestmt)
      match(:setstmt)
      match(:equalitystmt)
      match(:logicstmt)
      match(:mathstmt)
      match(:printstmt)
      match(:returnstmt)
      match(:arrayaddstmt)
      match(:func_call)
    end

    rule :arraygetstmt do
      # Get a value out of an array
      # For example: :array:[0] or :array:['a']

      match(/:\w+:\[\d+\]/) {|call|
        # Sets the "call" variable to the variable name
        # and the index the user requested
        call = call.split("[")
        call[0] = call[0][1..-2]
        call[1] = call[1][0..-2]

        # Gets the variable by name
        if (@@vars.has_key?(call[0]))
          var = @@vars[call[0]]
        else
          raise "The variable #{call[0]} does not exist"
        end

        # If the variable is not of the type array,
        # you can't get a value from it
        if (!var.type == "Array")
          raise "The variable #{var.name} is not of type array,
but is trying to be accessed as one"
        end

        # Checks so that you aren't trying to get a
        # value outside of the variables scope
        index = call[1].to_i
```

```
          if (index >= var.get_lowest_self.value.length)
            raise "The index #{index} is out of range from the
variable #{var.name}"
          end

          # Returns the value
          var.get_lowest_self.value[index] }
        match(/:\w+:\['\w+'\]/) {|call|
          # Sets the "call" variable to the variable name
          # and the variable name containing the index the user
requested
          call = call.split("[")
          call[0] = call[0][1..-2]
          call[1] = call[1][0..-2]

          # Gets the variable by name
          if (@@vars.has_key?(call[0]))
            var = @@vars[call[0]]
          else
            raise "The variable #{call[0]} does not exist"
          end

          # If the variable is not of the type array,
          # you can't get a value from it.
          if (!var.type == "Array")
            raise "The variable #{var.name} is not of type array,
but is trying to be accessed as one"
          end

          # Checks so that you aren't trying to get a
          # value outside of the variables scope
          index = set_array_content([call[1]])[0]
          if (index >= var.get_lowest_self.value.length)
            raise "The index #{index} is out of range from the
variable #{var.name}"
          end

          # Returns the value
          var.get_lowest_self.value[index] }
      end

      rule :arrayaddstmt do
        # Add a constant, a variable or a value
        # from another array to this array
        match(/:\w+:<</, :DEFAULT_CLASS) {|call, toadd|
          # Set the "call" variable to the name of the array
variable
          call = call.split(":")[1]

          # Gets the VarHolder object for the array variable.
          if (@@vars.has_key?(call))
            call = @@vars[call]
          else
            raise "The variable #{call} does not exist"
          end

          # Checks if the constant you are adding is an array,
          # and adds it specially. Otherwise adds it normally.
          if (toadd.class == String && toadd.is_array?)
            toadd = toadd[1..-2].gsub("_", "").split(",")
            call.get_lowest_self.value << set_array_content(toadd)
          else
            call.get_lowest_self.value << toadd
          end }
        match(/:\w+:<</, :GETVAR) {|call, toadd|
          # If the gotten variable is empty, the value can not be
appended
          if (toadd == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end

          # Set the "call" variable to the name of the array
variable
          call = call.split(":")[1]

          # Gets the VarHolder object for the array variable.
          if (@@vars.has_key?(call))
            call = @@vars[call]
          else
            raise "The variable #{call} does not exist"
          end
          # Adds the value to the array
          call.get_lowest_self.value << toadd }
        match(/:\w+:<</, :arraygetstmt) {|call, toadd|
          # Set the "call" variable to the name of the array
variable
          call = call.split(":")[1]

          # Gets the VarHolder object for the array variable.
          if (@@vars.has_key?(call))
            call = @@vars[call]
          else
            raise "The variable #{call} does not exist"
          end

          # Adds the value to the array
          call.get_lowest_self.value << toadd }
      end

      rule :ifstmt do
        # Matches and executes if-statment
        match('if', :DIGIT) {|_, index|

          # Creates a scope
          var_backup = @@vars.dup

          return_value = index

          # Checks if statement's expression is true
          if (@@parser.parse(@@ifstmts[index].logicstmt))

            # Loops through all the content in the if statement.
            # If the content is a VarHolder, it adds that
            # variable to the @@vars Hash, otherwise it parses the
line
            # If the line contains a return statement, the loop
breaks
            @@ifstmts[index].content.each do |line|
              if @@returnstmt
                @@returnstmt = false
                break
              elsif (line.class == VarHolder)
                @@vars[line.name] = line
              else
                return_value = @@parser.parse(line)
              end
            end

          # Checks if the statements has any otherwise content
          elsif (!@@ifstmts[index].othercont.empty?)

            # Loops through all the content in the otherwise
statement.
            # If the content is a VarHolder, it adds that
            # variable to the @@vars Hash, otherwise it parses the
line
            # If the line contains a return statement, the loop
breaks
            @@ifstmts[index].othercont.each do |line|
              if @@returnstmt
                @@returnstmt = false
                break
              elsif (line.class == VarHolder)
                @@vars[line.name] = line
              else
                return_value = @@parser.parse(line)
              end
            end
          end

          # Resets scope
          @@vars = var_backup
          return_value }
      end

      rule :whilestmt do
        # Mathes while statements
        match('while', :DIGIT) {|_, index|
          var_backup = @@vars

          return_value = index
          # Checks if the statement's expression is true
          while(@@parser.parse(@@whilestmts[index].logicstmt))

            # Loops through all the content in the while statement.
            # If the content is a VarHolder, it adds that
            # variable to the @@vars Hash, otherwise it parses the
line
            # If the line contains a return statement, the loop
breaks
            @@whilestmts[index].content.each do |line|
              if @@returnstmt
                @@returnstmt = false
                break
              elsif (line.class == VarHolder)
                @@vars[entry.name] = entry
              else
                return_value = @@parser.parse(line)
              end
            end
          end

          # Resets scope
          @@vars = var_backup
          return_value }
      end

      rule :setstmt do
        # Matches all instances where a variable gets a value
        match(:USEVAR, '=', :arraygetstmt) {|var, _, val|
          # Matches setting a variable to a value from an array
          var = @@vars[var[1..-2]]
          # Checks if the value is of the same class as the
variable
          if (var.type == val.class.name ||
              (var.type == "Boolean" && !!val == val) ||
              (var.type == "Integer" && val.is_a?(Integer)))
            var.get_lowest_self.value = val
          else
            raise "The variable #{var.name} is being set to wrong
type!\nError: :#{var.name}:=#{val}"
          end }

        match(:USEVAR, '=', @@arrayRegex) {|var, _, val|
          # Matches setting a variable to an array constant
          var = var[1..-2]
          val = val[1..-2]
          # Checks if the variable is an Array
          if (@@vars[var].type == "Array")
            # Sets the array's content to the correct type.
            # Sets the value to the new array and sets length
            # to the length of the new array
            array = set_array_content(val.split(","))
            @@vars[var].get_lowest_self.value = array
            if (@@vars[var] == @@vars[var].get_lowest_self)
              @@vars["length"].value = array.length
            else
              @@vars[var].value["length"].value = array.length
            end
          else
            raise "The variable #{var} is being set to wrong
type!\nError: :#{var}:=[#{val}]"
          end }

        match(:USEVAR, '=', /"[^"]*"/) {|var, _, val|
          # Matches setting a variable to a string constant
          var = var[1..-2]
          val = val[1..-2]
          # Checks if the variable is a String
          if (@@vars[var].type == "String")
            @@vars[var].get_lowest_self.value = val
          else
            raise "The variable #{var} is being set to wrong
type!\nError: :#{var}:=\"#{val}\""
          end }

        match(:USEVAR, '=', :mathstmt) {|var, _, val|
          # Matches setting a variable to the value of a math
statement
          var = var[1..-2]

          # Checks if both variables are either Float or Integer,
          # the only possible types math statements can be
          if (val.class == Float && @@vars[var].type == "Float")
            @@vars[var].get_lowest_self.value = val
          elsif (val.is_a?(Integer) && @@vars[var].type ==
"Integer")
            @@vars[var].get_lowest_self.value = val
          elsif (val.class.name == @@vars[var].type ||
@@vars[var].type == "Boolean" && !!val == val)
            @@vars[var].get_lowest_self.value = val
          else
            raise "The variable #{var} is being set to wrong
type!\nError: :#{var}:=#{val}"
          end }

        match(:USEVAR, '=', :logicstmt) {|var, _, val|
          # Matches setting a variable to the value of a logic
statements
          var = var[1..-2]

          # Checks if the variable is a Boolean,
          # the only possible type logic statements can be
          if (@@vars[var].type == "Boolean")
            @@vars[var].get_lowest_self.value = val
          else
            raise "The variable #{var} is being set to wrong
type!\nError: :#{var}:=#{val}"
          end }

        match(:USEVAR, '=', :GETVAR) {|var, _, val|
          # If the gotten variable is empty, the value can not be
appended
          if (val.value == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end
          # Matches setting a variable to the value of another
variable
          var = var[1..-2]
          # Checks if both variables are arrays and sets the length
variable,
          # if that is the case
          if (val.type == "Array" && @@vars[var].type == "Array")
            if (@@vars[var].value.class != Hash)
              @@vars["length"].value = val.value.length
            else
              if (!@@vars[var].value.has_key?("length"))
                @@vars[var].value["length"] =
VarHolder.new("length", "Integer")
              end
              @@vars[var].value["length"].value = val.value.length
            end
          end

          # Checks if both variables are of the same type
          if (val.type == @@vars[var].type)
            @@vars[var].get_lowest_self.value = val.value
          else
            nil
          end }

        match(:USEVAR, '=', :func_call) {|var, _, val|
          # Matches setting a variable to the return value of a
function call
          var = var[1..-2]

          # Checks if the value is of the same class as the
variable
          if (@@vars[var].type == val.class.name ||
              (@@vars[var].type == "Boolean" && !!val == val) ||
              (@@vars[var].type == "Integer" && val.is_a?
(Integer)))
            @@vars[var].get_lowest_self.value = val
          else
            raise "You are trying to set a variable (#{var}) to the
wrong type!"
          end }
      end

      rule :mathstmt do
        # Math statement matches down through the different
mathematical
        # expressions down to "mathexpr". This is to ensure the
correct
```

```
        # priorities of the expressions
        match(:plusstmt)
      end

      rule :plusstmt do
        match(:plusstmt, '+', :minusstmt) {|e, _, t| e + t }
        match(:minusstmt)
      end

      rule :minusstmt do
        match(:minusstmt, '-', :multstmt) {|m, _, t| m - t }
        match(:multstmt)
      end

      rule :multstmt do
        match(:multstmt, '*', :divstmt) {|t, _, q| t * q }
        match(:divstmt)
      end

      rule :divstmt do
        match(:divstmt, '/', :modstmt) {|a, _, b| a / b }
        match(:modstmt)
      end

      rule :modstmt do
        match(:modstmt, '%', :powstmt) {|a, _, b| a % b }
        match(:powstmt)
      end

      rule :powstmt do
        match(:powstmt, '**', :mathexpr) {|f, _, q| f ** q }
        match(:mathexpr)
      end

      rule :mathexpr do
        match('(', :mathstmt, ')') {|_, e, _| e }
        match(:GETVAR) {|a|
          # If the gotten variable is empty, the value can not be
appended
          if (a.value == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end
          a.value }
        match(:DIGIT)
      end

      rule :logicstmt do
        # Logic statement matches down through the different
logical
        # expressions down to "logicexpr". This is to ensure the
correct
        # priorities of the expressions
        match(:orstmt)
      end

      rule :orstmt do
        match(:orstmt, '||', :andstmt) {|a, _, b| a || b }
        match(:andstmt)
      end

      rule :andstmt do
        match(:andstmt, '&&', :notstmt) {|a, _, b| a && b }
        match(:notstmt)
      end

      rule :notstmt do
        match('!', :notstmt) {|_, a| !a }
        match(:logicexpr)
      end

      rule :logicexpr do
        match('(', :logicstmt, ')') {|_, l, _| l }
        match('true') { true }
        match('false') { false }
        match(:GETVAR) {|a|
          # If the gotten variable is empty, the value can not be
appended
          if (a.value == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end
          a.value }
      end

      rule :equalitystmt do
        # Equality statement is used for testing equalities,
        # such as ==, < and >
        match(:equalitystmt, '<', :equalitystmt) {|a, _, b| a < b }
        match(:equalitystmt, '>', :equalitystmt) {|a, _, b| a > b }
        match(:equalitystmt, '==', :equalitystmt) {|a, _, b| a == b
}
        match('(', :equalitystmt, ')') {|_, a, _| a }
        match(:DEFAULT_CLASS) {|a|
          if (a.class == String && a.is_array?)
            set_array_content(a[1..-2].split(","))
          else
            a
          end }
        match(:GETVAR) {|a|
          # If the gotten variable is empty, the value can not be
appended
          if (a.value == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end
          a.value }
        match(:mathexpr)
        match(:logicexpr)
      end

      def print_array(array)
        # Prints the contents of an array. If the array contains
another array,
        # the function prints it recursively
        print "["
        array.length.times do |i|
          entry = array[i]
          if (entry.class == Array)
            print_array(entry)
          elsif (entry.class == String)
            entry = '"' + entry unless (entry[0] == '"')
            entry = entry + '"' unless (entry[-1] == '"')
            print entry.gsub("\\s", " ")
          else
            print entry
          end
          print ", " unless (i == array.length - 1)
        end
        print "]"
      end

      rule :printstmt do
        # Matches printing a value from an array
        match('print', :arraygetstmt) {|_, a|
          # Checks if the value is a String or an Array
          # and prints it specially, otherwise normally
          if (a.class == String)
            print a.gsub("\\n", "\n").gsub("\\s", "\s");
          elsif (a.class == Array)
            print_array(a)
          else
            print a
          end
          a }

        match('print', :DEFAULT_CLASS) {|_, a|
          # Matches printing a default class constant
          if (a.class == String)
            # If a is a String, it can be either an Array or a
String
            if (a.is_array?)
              a = a[1..-2].split(",")
              a = set_array_content(a)
              print_array(a)
            else
              print a.gsub("\\n", "\n").gsub("\\s", " ")
            end
          elsif (a.class == Array)
            print_array(a)
          else
            print a
          end
          a }
        match('print', :GETVAR) {|_, a|
          # If the gotten variable is empty, the value can not be
appended
          if (a.value == nil)
            raise "You are trying to get the value of a variable
that has not yet been given a value!"
          end
          # Matches printing the value of a variable
          a = a.get_lowest_self.value
          # Checks if the value is a String or an Array
          # and prints it specially, otherwise normally
          if (a.class == String)
            print a.gsub("\\n", "\n").gsub("\\s", " ")
          elsif (a.class == Array)
            print_array(a)
          else
            print a
          end
          a }

        # Matches printing the return value of a function call
        match('print', :func_call) {|_, a| print a; a }
      end

      rule :returnstmt do
        # Matches all return statements
        match('return', :DEFAULT_CLASS) {|_, a|
          # Return default class constant
          # Check if "a" is an array, and set content if that is
the case
          # Check if the returntype is the correct type.
          @@returnstmt = true
          if (a.class == String && a.is_array?)
            a = set_array_content(a[1..-2].split(","))
          end

          if (@@returntype == a.class.name ||
              (@@returntype == "Boolean" && !!a == a) ||
              (@@returntype == "Integer" && a.is_a?(Integer)) ||
              (@@returntype == "ARRAYGETSTMT"))
            a
          else
            raise "Trying to return a variable (#{a}) that is not
the correct class"
          end }

        match('return', :equalitystmt) {|_, a|
          # Return an equality statement
          # Check if "a" is the correct type (Boolean),
          # and that the returntype is set to Boolean
          @@returnstmt = true
          if (@@returntype == "Boolean" && !!a == a)
            a
          else
            raise "Trying to return a variable (#{a}) that is not
the correct class"
          end }

        match('return', :arraygetstmt) {|_, a|
          # Return a value from an array
          # Check if the returntype is of the same type as "a"
          # If the returntype is "ARRAYGETSTMT", you can return
anything
          @@returnstmt = true
          if (@@returntype == a.class.name ||
              (@@returntype == "Boolean" && !!a == a) ||
              (@@returntype == "Integer" && a.is_a?(Integer)) ||
              (@@returntype == "ARRAYGETSTMT"))
            a
          else
            raise "Trying to return a variable (#{a}) that is not
the correct class"
          end }

        match('return', :func_call) {|_, a|
          # Return the return value from a function
          # Check if the returntype is of the same type as "a"
          # If the returntype is "ARRAYGETSTMT", you can return
anything
          @@returnstmt = true
          if (@@returntype == a.class.name ||
              (@@returntype == "Boolean" && !!a == a) ||
              (@@returntype == "Integer" && a.is_a?(Integer)) ||
              (@@returntype == "ARRAYGETSTMT"))
            a
          else
            raise "Trying to return a variable (#{a}) that is not
the correct class"
          end }
      end

      def memfunc_call(func_name)
        # This function is called if you make a function call
containing two ":"
        var_name = get_chars_between_char(':', func_name)[0]
        if (var_name == nil)
          # If "var_name" is empty, it means we are trying
          # to access a member function from within another member
function
          # For example: set :: to 5
          vars_backup = @@vars.dup

          return_val = func_call(func_name, true)

          @@vars = vars_backup
          return return_val
        end
        var = @@vars[var_name]

        # If the gotten variable is empty, the value can not be
appended
        if (var.value == nil)
          raise "You are trying to get the value of a variable that
has not yet been given a value!"
        end

        # Get the ClassHolder for the type of the variable
        class_holder = nil

        if (@@classes.has_key?(var.type))
          class_holder = @@classes[var.type].dup
        elsif (@@defaultclasses.has_key?(var.type))
```

```
  class_holder = @@defaultclasses[var.type].dup
end

# Create a new scope
func_backup = @@funcs.dup
var_backup = @@vars.dup
@@funcs = @@funcs.merge(class_holder.memfuncs)
@@vars = class_holder.memvars.dup

if (var.value.class == Hash)
  var.value.each do |key, val|
    @@vars[key].value = val.value
  end
end

# Set all the parameters to the value they had before.
get_chars_between_char("'", func_name).each do |var_name|
  @@vars[var_name] = var_backup[var_name].dup
end

# Run the function
return_val = func_call(func_name, true)

# Save the changed variables
vars_changes = @@vars

# Revert to the previous scope
@@funcs = func_backup
@@vars = var_backup

# Set the value of the called upon variable to the correct
value(s)
if (@@vars[var.name].value == nil)
  @@vars[var.name].value = {}
  class_holder.memvars.each do |key, val|
    @@vars[var.name].value[key] = val.dup
  end
elsif (@@vars[var.name].value.class == Hash)
  @@vars[var.name].value.each do |key, val|
    @@vars[var.name].value[key] = vars_changes[key].dup
  end
```

```
        end

        return return_val
      end

      def func_call(func_name, memfunc_call=false)
        # This function is called whenever a function call is made.
        # Start by getting all the possible function names
        func = func_name.get_possible_function_name
        real_func_name = nil
        # Loop through the names, and try to find the correct one
        func.each do |name|
          if @@funcs.has_key?(name)
            func = @@funcs[name]
            real_func_name = name
            break
          elsif @@defaultfuncs.has_key?(name)
            func = @@defaultfuncs[name]
            real_func_name = name
            break
          end
        end

        # If no function was found, the variable func will be of
type Array
        if (func.class == Array)
          raise "The function call for #{func_name} was not found!"
        end

        # If the function does not exist in the current class,
        # "func" will be of type string, containing the name
        # of the class which will have the function
        class_name = func
        while (func.class == String)
          if (@@classes.has_key?(func))
            class_name = func
            func = @@classes[func].memfuncs[real_func_name].dup
          elsif (@@defaultclasses.has_key?(func))
            class_name = func
            func =
@@defaultclasses[func].memfuncs[real_func_name].dup
```

```
          end
        end

        # Now that the function is found, get all the variables
        # from the function call, and loop through them
        # to set the parameters to the correct values

        var_array = func_name.get_vars_in_order(func.general_name)
        var_array.length.times do |i|
          # If the value of the parameter hasn't been set,
          # set it to the value it should have.
          if (func.parameters[i].value == nil)
            func.parameters[i].value = {}
            if (@@classes.has_key?(func.parameters[i].type))
              @@classes[func.parameters[i].type].memvars.each do |
key, val|
                func.parameters[i].value[key] = val.dup
              end
            elsif (@@defaultclasses.has_key?
(func.parameters[i].type))

@@defaultclasses[func.parameters[i].type].memvars.each do |key,
val|
                func.parameters[i].value[key] = val.dup
              end
            end
          end
          var = var_array[i]
          # If "var" is a variable
          if ((var[0] == "'" && var[-1] == "'") || (var[0] == ":"
&& var[-1] == ":"))
            var = var[1..-2]
            # If "var" is the user trying to get a value from an
array
            # For example: function ':array:[0]';
            if (var.include?("[") && var.include?("]"))
              # Get the array's name and the index
              split = var.split("[")
              split[0] = split[0][1..-2]
              split[1] = split[1][0..-2]
              # Get the value of that index in the array variable
              val =
@@vars[split[0]].get_lowest_self.value[split[1].to_i]
              # Check if the parameter type and the variable
              # type are of the same type, and add the variable in
that case
              if (func.parameters[i].type == "String" && val.class
== String)
                func.parameters[i].get_lowest_self.value = val
              elsif (func.parameters[i].type == "Integer" &&
val.is_a?(Integer))
                func.parameters[i].get_lowest_self.value = val
              elsif (func.parameters[i].type == "Boolean" && !!val
== val)
                func.parameters[i].get_lowest_self.value = val
              elsif (func.parameters[i].type == "Float" &&
val.class == Float)
                func.parameters[i].get_lowest_self.value = val
              elsif (func.parameters[i].type == "Array" &&
val.class == Array)
                func.parameters[i].get_lowest_self.value = val
                if (func.parameters[i].value.class == Hash)
                  func.parameters[i].value["length"] =
VarHolder.new("length", "Integer")
                  func.parameters[i].value["length"].value =
val.length
                end
              else
                raise "The parameter '#{var}' is not of the correct
type!"
              end
            # Check that the variable is of the same type as the
parameter
            elsif (func.parameters[i].type ==
@@vars[var].get_lowest_self.type)
              func.parameters[i].get_lowest_self.value =
@@vars[var].get_lowest_self.value
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          # "var" is an array
          elsif (var.is_array?)
            if (func.parameters[i].type == "Array")
              var = var[1..-2].gsub("_", "").gsub("\\s", " ")
              func.parameters[i].get_lowest_self.value =
set_array_content(var.split(","))
              if (func.parameters[i].value.class == Hash)
                func.parameters[i].value["length"] =
VarHolder.new("length", "Integer")
                func.parameters[i].value["length"].value =
func.parameters[i].get_lowest_self.value.length
              end
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          # "var" is an int
          elsif (var.is_int?)
            if (func.parameters[i].type == "Integer")
              func.parameters[i].get_lowest_self.value = var.to_i
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          # "var" is a float
          elsif (var.is_float?)
            if (func.parameters[i].type == "Float")
              func.parameters[i].get_lowest_self.value = var.to_f
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          # "var" is a string
          elsif (var.is_string?)
            if (func.parameters[i].type == "String")
              func.parameters[i].get_lowest_self.value = var[1..-
2].gsub("\\s", " ")
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          # "var" is a boolean
          elsif (var.is_boolean?)
            if (func.parameters[i].type == "Boolean")
              func.parameters[i].get_lowest_self.value = var.to_b
            else
              raise "The parameter '#{var}' is not of the correct
type!"
            end
          end
        end

        if (@@vars["self"].class == VarHolder &&
            @@vars["self"].type == "Array" &&
            @@vars["self"].value != nil &&
            @@vars["length"].class == VarHolder &&
            @@vars["length"].value == nil)
          @@vars["length"].value = @@vars["self"].value.length
        end

        # Create a new scope
        vars_backup = @@vars.dup
        # Member functions should be able to use
        # member variables, but functions should not
        # be able to use variables outside its scope
        @@vars = {} unless memfunc_call

        # Add all the parameters to the scope
        func.parameters.each do |param|
          @@vars[param.name] = param
        end

        # If the function was inherited, get all the
        # variables from that class and put it in the scope
        if (class_name.class == String)
          if (@@classes.has_key?(class_name))
            @@classes[class_name].memvars.each do |key, val|
              @@vars[key] = val
            end
          else
            @@defaultclasses[class_name].memvars.each do |key, val|
              @@vars[key] = val
            end
```

```
          end
        end

        # Backup the previous return type
        returntype_backup = @@returntype
        @@returntype = func.returntype

        return_val = nil
        # Parse the contents of the function
        func.contents.each do |entry|
          if @@returnstmt
            break
          elsif (entry.class == VarHolder)
            @@vars[entry.name] = entry
          else
            return_val = @@parser.parse(entry)
          end
        end

        # Restore the previous scope
        @@returntype = returntype_backup

        @@returnstmt = false

        @@vars = vars_backup

        return return_val
      end

      rule :func_call do
        # Matches all function calls.
        # If it contains two ":", it is a member function call
        # If it does not, it is a normal function call
        match(/[^;{}]+/) {|a|
          if (a =~ /:\w+:/ || a =~ /::/) # Member function call
            memfunc_call(a)
          else # Normal function call
            func_call(a)
          end }
      end
      rule :DEFAULT_CLASS do
        # Matches all default classes:
        #   Array
        #   String
        #   Integer/Float
        #   Boolean
        match(@@arrayRegex) {|a| #Array
          if (a.is_array?)
            a
          else
            nil
          end }
        match(/"[^"]*"/) {|a|
          if (a.is_string?)
              a[1..-2]
          else
            nil
          end } # String
        match(:mathstmt) # All mathematicals
        match(:logicstmt) # All boolean logic
      end

      rule :GETVAR do
        # Matches the getting of the value of a variable.
        # Checks if the variable exists, and returns the VarHolder.
        match(/'\w+'/) {|a|
          a = a[1..-2]
          if @@vars.has_key?(a)
            var = @@vars[a].get_lowest_self
            if (var.value == nil)
              raise "You are trying to get the value of a variable
that has not yet been given a value!"
            else
              var
            end
          else
            nil
          end}
      end

      rule :USEVAR do
        # Matches the using of a variable.
        match(/:\w+:/)
      end

      rule :DIGIT do
        # Matches integers and floats
        match(/\d+\.\d+/) {|float|
          if (float.is_float?)
            float.to_f
          else
            nil
          end }
        match(/\d+/) {|integer|
          if (integer.is_int?)
            integer.to_i
          else
            nil
          end }
      end

    end
  end

  def read_file(fileloc)
    # Make sure that no file is read more than once
    @@includesdone << fileloc
    # Read the file
    file_to_superstring(fileloc).each do |line|
      if (line =~ /use \".+\.etr\";/)
        line = line[/\".+\.etr\"/][1..-2]
        read_file(line) unless (@@includesdone.include?(line))
      else
        @reader.parse(line)
      end
    end

    # All default classes and functions should be accessible from
    # any scope, so they get their own variables.
    if (fileloc == "__DEFAULTTHINGS__.etr")
      @@defaultfuncs = @@funcs
      @@funcs = {}
    @@defaultclasses = @@classes
    @@classes = {}
  end
end

def file_to_superstring(fileloc)
  # Takes a file location, adds all the
  # lines to one long string besides
  # include statements, which are set
  # separately
  return_val = [""]
  File.open(fileloc).each_line do |line|
    if (line =~ /use \".+\.etr\";/)
      return_val << line
      return_val << ""
    else
      line.delete!("\n")
      return_val[return_val.length-1] += line
    end
  end
  return return_val
end

def parse(fileloc)
  # Parses the file by first reading it
  # and then parsing it
  read_file(fileloc)

  # If you are in debug-mode, print out all the saved data
  if (ARGV.length > 1)
    puts "******** CLASSES ***********"
    @@defaultclasses.each do |key, val|
      print_class(val)
      puts
    end
    @@classes.each do |key, val|
      print_class(val)
      puts
    end
    puts "******** MAIN FUNCTIONS ***********"
```

```
    @@defaultfuncs.each do |key, val|
      print_func(val)
      puts
    end
    @@funcs.each do |key, val|
      print_func(val)
      puts
    end

    puts "******** MAIN VARIABLES ***********"
    @@vars.each do |key, val|
      print_var(val)
      puts
    end

    puts "******** READ FILES ***********"
    @@includesdone.each do |include|
      puts include
    end

    puts "******** MAIN CONTENT ***********"
    @@maincontent.each do |entry|
      puts entry
    end

    puts "******** IFS ***********"
    @@ifstmts.length.times do |i|
      print "#{i} => "
      print [@@ifstmts[i]]
      puts
    end

    puts "******** WHILES ***********"
    @@whilestmts.length.times do |i|
      print "#{i} => "
      print [@@whilestmts[i]]
      puts
    end

    puts "\n\n\n"
    puts "******** INTERPRET ***********"
    end
    returnval = ""
    @@maincontent.each do |line|
      if @@returnstmt
        return returnval
      else
        returnval = @@parser.parse(line)
      end
    end
  end

  def log(state = true)
    # Sets the state of the loggers
    if (state)
      @reader.logger.level = Logger::DEBUG
      @@parser.logger.level = Logger::DEBUG
    else
      @reader.logger.level = Logger::WARN
      @@parser.logger.level = Logger::WARN
    end
  end

  def print_class(ch)
    # Prints a ClassHolder in a coherent way
    puts "NAME: "
    puts ch.name
    puts "MEMFUNCS: "
    ch.memfuncs.each do |key, val|
      print key
      puts
      if (val.class == String)
        puts "  can be found in #{val}"
        next
      end
      print "  PARAM: "
      print val.parameters
      puts
      print "  CONT: "
      print val.contents
      puts
      print "  RET: "
      print val.returntype
      puts
    end
    puts
    puts "MEMVARS: "
    print ch.memvars
    puts
    puts "INHERITS: "
    print ch.inherits
    puts
  end

  def print_func(fh)
    # Prints a FuncHolder in a coherent way
    print fh.name
    print ":\n"
    print "  PARAMETERS: "
    print fh.parameters
    puts
    print "  CONTENT: "
    print fh.contents
    puts
    print "  RETURN TYPE: "
    print fh.returntype
    puts
  end

  def print_var(vh)
    # Prints a VarHolder in a coherent way
    print vh.name
    print ":\n"
    print "  CLASS: "
    puts vh.type
    print "  VALUE: "
    puts vh.value
  end
end

# Initialize the parser
etr = ETR.new
# If more than just the file name is inserted, enter debug mode
```

```
if (ARGV.length > 1)
  etr.log(true)
  $stdout.reopen(File.new("./DEBUG.txt", 'w'))
else
  etr.log(false)
end
# Check that the user inserted a filename, and that it is of the
correct type
if (ARGV.length > 0)
  if (ARGV[0][-4..-1] != ".etr")
    raise "The file given is not of the correct type!"
  end
else
  raise "No file given"
end
returnval = nil

# Parse the file containing all the default content
etr.read_file("__DEFAULTTHINGS__.etr")
# Parse the file the user requested.
begin
  returnval = etr.parse("#{ARGV[0]}")
rescue Exception => ex
  puts ex.message
end

returnval
```

5.2 nonlanguagethings.rb

```
# -*- coding: utf-8 -*-
def get_vars_from_func_call(call)
  # Returns list of all variables used in a function call
  array = []

  if (call.count_chars_in_row("'").max > 1 ||
call.count_chars_in_row(":").max > 1)
    raise "You can only include one variable in a function
call.\nError: #{call}"
  end

  array = get_chars_between_char("'", call, array)
  array = get_chars_between_char(":", call, array)

  return array
end

def get_chars_between_char(char, str, array=[])
  # Returns list of characters between "char"
  while (str.count(char) >= 2)
    str = str[str.index(char)+1..-1]
    var = str[0..str.index(char)-1]
    # Checks if there are more characters to extract
    if (var.include?("'") || var.include?(":"))
      array << get_vars_from_func_call(var)
      array.flatten!
    else
      array << var if (!array.include?(var))
    end
    str = str[str.index(char)+1..-1]
  end
  return array
end

class Array
  def delete_first_of(char)
    # Removes the first instance of "char" from the array
    self.delete_at(self.index(char) || self.length)
  end
end

class String
  def to_b
    # Converts "self" to Boolean
    self == "true"
  end

  def is_digit?
```

```
    # Returns true if "self" is int or float
    (self.is_int? || self.is_float?)
  end

  def is_int?
    # Returns true if "self" is an Integer.
    # The function tries to convert "self" to an Integer,
    # if it's possible the function returns true.
    true if (Integer(self)) rescue false
  end

  def is_float?
    # Returns true if "self" is a Float.
    # The function tries to convert "self" to a Float,
    # if it's possible the function returns true.
    true if (Float(self)) rescue false
  end

  def is_boolean?
    # Returns true if "self" is "true" or "false"
    return true if (self == "true" || self == "false")
    return false
  end

  def is_string?
    # Returns true if both the first and the last character in
"self" is '"'
    return true if (self[0] == '"' && self[-1] == '"')
    return false
  end

  def is_array?
    # Returns true if the first character in "self" is "[" and the
last "]"
    return true if (self[0] == "[" && self[-1] == "]")
    return false
  end

  def is_math_expr?
    # Returns true if "self" contains any mathematical operators
    return true if (self =~ /(\*\*|\+|-|\/|\*|%)/)
    return false
  end

  def is_logic_expr?
    # Returns true if "self" contains any logical operators
    return true if (self =~ /(==|<|>|=|!|&&|\|\|)/)
    return false
  end

  def get_return_type(vars={})
    # Returns correct type of the string
    return "Array" if self.is_array?
    return "String" if self.is_string?
    return "Integer" if (self.is_int? || self.is_math_expr?)
    return "Float" if self.is_float?
    return "Boolean" if (self.is_boolean? || self.is_logic_expr?)
    return "ARRAYGETSTMT" if (self =~ /(:\w+:\['\w+'\]|:\w+:\
[\d+\])/)
    return vars[self].type if (vars.has_key?(self))
    return nil
  end

  def is_default_class?
    # Returns true if the string is a default class
    return true if (self.is_boolean? || self.is_digit? ||
self.is_string? || self.is_array?)
    return false
  end

  def count_chars_in_row(char)
    # Returns list of the number of times "char" is found in a row
in "self"
    return [0] if (!self.include?(char))

    array = []
    copy = self
    # Checks if the string has more instances of "char"
    while (copy.include?(char))
      count = 1
      index = copy.index(char)
    # Checks if the index is inside the length and
    # if the next character in the string matches "char"
    while (copy.length > index && copy[index+1] == char)
      count += 1
      index += 1
    end

    copy = copy[index+1..-1]
    array << count
  end

  return array
end

def calc_values(mod, length, name)
  # Returns an array containing whether or not
  # a number should be in the function name
  # for all possible values (depending on "length" and "mod")
  length = 2 ** length
  counter = 0
  parameter = false
  return_array = []

  while (counter < length)
    # For every mod, switch whether or not a number should be
    # itself or "$"
    mod.times do |i|
      if (parameter)
        return_array << name
      else
        return_array << "$"
      end
      counter += 1
    end
    parameter = !parameter
  end

  return return_array
end

def calc_all_values(array, str)
```

```
    # Makes use of the fact that the numbers being in the name
follow a pattern
    # of doubling the window of being in the name or not.
    # For example:
    #     If we have a function name (func_$1_$2), a possible call
could be:
    #     func 1 2. However, there could be a function named
func_1_$1.
    #     So the fact that the number is in the name or not follows
a pattern:
    #     1 : in, not in, in, not in
    #     2 : in, in, not in, not in
    # This function uses the above logic to determine all
possibilities of
    # a number being in the function name or not.
    mod = 1
    vals = {}

    #
    array.each do |i|
      vals[i] = calc_values(mod, array.length, str[i])
      mod *= 2
    end

    return_array = []
    (2 ** array.length).times do |i|
      temp = {}
      array.each do |j|
        temp[j] = vals[j][i]
      end
      return_array << temp
    end

    return return_array
  end

  def get_possible_function_name
    # Returns list of all possible function names
    func = self.gsub(" ", "_")

    # In the function name, replaces the name of the object the
    # function is being called from (specified between ":") with
"$n"
    # For example: :a: is array?; => $n_is_array?
    array = get_chars_between_char(":", func)
    array.each do |elem|
      elem = ":" + elem + ":"
      func = func.gsub(elem, "$n")
    end
    func = func.gsub("::", "$n") if (array.empty?)

    # In the function name, replaces parameter names
    # (specified between "'") with "$"
    # For example: multiply 'x' by 'y'; => multiply_$_by_$
    array = get_chars_between_char("'", func)
    (1..array.length).each do |i|
      elem = "'" + array[i-1] + "'"
      func = func.sub(elem, "$")
    end

    # Replaces strings in the function call with "$"
    while (func.count('"') > 1 && func.count_chars_in_row('"') !=
2)
      str = func[func.index('"')+1..-1]
      str = str[0..str.index('"')-1]
      str = '"' + str + '"'
      str = '""' if (str == '""""')
      func = func.sub(str, "$")
    end

    # Replaces floats, arrays and boolean values with "$"
    func = func.gsub(/\d+\.\d+/, "$") # Float
    func = func.gsub(/\[[[^\]],]*[^\]]*\]/, "$") # Array
    func = func.gsub(/(true|false)/, "$")

    # Adds all numbers in "func" to ----------------
    array = []
    all_digits = []
    func.split("_").each do |entry|
      if (entry.is_digit?)
        all_digits << entry
      end
```

```
    end

    all_values = calc_all_values(all_digits, func)

    # Calculates all the possible function names depending on the
    # values from "calc_all_values" and adds them to an array
    all_values.each do |entry|
      temp_func = func.dup
      entry.each do |key, val|
        temp_func[key] = val
      end
      array << temp_func
    end

    return array
  end

  def special_split(splitchar)
    # Splits "self" on all "splitchar", if "splitchar" isn't
between
    # ":", "'", "[" or "]". Returns array containing all substrings
    return_array = []
    copy = self.dup
    start = 0
    finish = 0

    getvar = false
    usevar = false
    string = false
    array = false

    while (finish < copy.length)
      case copy[finish]
      when ":"
        usevar = !usevar
      when "'"
        getvar = !getvar
      when '"'
        string = !string
      when "["
        array = true
      when "]"
        array = false
      when splitchar
        if (!getvar && !usevar && !string && !array)
          return_array << copy[start..finish-1]
          start = finish + 1
        end
      end
      finish += 1
    end

    return_array << copy[start..-1]
    return return_array
  end

  def get_vars_in_order(function_name)
    # Gets all the variables inserted to a function in order,
    # to be inserted in the correct order as parameters
    array = []

    # Split both the function name and the function call ("self")
by "_"
    funcsplit = function_name.special_split("_")
    selfsplit = self.special_split("_")

    # If the function name has "$" at an index,
    # add the same index to the returning array
    funcsplit.length.times do |i|
      if (funcsplit[i] == "$")
        array << selfsplit[i]
      end
    end

    return array
  end
end

class ClassHolder
  # Holder for all classes.
  # "name" is the class name, "memfuncs" contains the member
functions,
  # "memvars" contains the member variables and
  # "inherits" contains the class names of all the classes
  # the class inherits from
  attr_accessor :name, :inherits, :memfuncs, :memvars
  def initialize()
    @name = nil
    @memfuncs = {}
    @memvars = {}
    @inherits = []
  end
end

class VarHolder
  # Holder for all variables.
  # "name" is the variable name, "type" is the variable type and
  # "value" stores the current value of the variable
  attr_accessor :name, :type, :value
  def initialize(name=nil, type=nil, value=nil)
    @name = name
    @type = type
    @value = value
  end

  def dup
    # Returns a copy of "self"
    return VarHolder.new(@name, @type, @value)
  end

  def get_lowest_self
    # Returns the deepest self
    val = self
    # While the value of val is of the type Hash,
    # set "val" to the next "self"
    while (val.value.class == Hash)
      if (val.value.has_key?("self"))
        val = val.value["self"]
      else
        raise "Can't find the deepest self for variable #{@name}"
      end
    end
    return val
  end
end

class FuncHolder
  # Holder for all functions.
  # "name" is the name of the function,
  # "parameters" contains all parameters of the function,
  # "contents" contains the content of the function and
  # "returntype" contains the type of the value the function
returns
  attr_reader :name
  attr_accessor :parameters, :contents, :returntype
  def initialize(name=nil, parameters=[], contents=[],
returntype=nil)
    @name = name
    @parameters = parameters
    @contents = contents
    @returntype = returntype
    check_name unless @name == nil
  end

  def content
    @contents
  end

  def name=(name)
    @name = name
    check_name
  end

  def check_name
    # Checks if the parameters in the function name is written in
the
    # correct order.
    return nil if @parameters.length == 0
    array = @name.split("$")
    expected = 1
    (1..array.length-1).each do |i|
      if (array[i][0] == expected.to_s)
        expected += 1
      elsif (array[i][0] == "n")
        expected += 0
      else
        raise "A function name has to have it's parameters in
order!\nError: #{@name}"
      end
    end
  end

  def find_return_in_holder(holder, ifs, whiles)
    # Returns the return statement of the function
    return_stmt = nil

    # Goes through each line in the functions content and sets
"return_stmt"
    # to the line if the line contains "return ". Also checks for
    # return statements in if and while statements recursively
    holder.content.each do |content|
      if (content.class == String && content =~ /return\s/)
        return_stmt = content
        break
      elsif (content.class == String && content =~ /if \d+/)
        return_stmt =
find_return_in_holder(ifs[content[/\d+/].to_i], ifs, whiles)
        break if (return_stmt != nil)
      elsif (content.class == String && content =~ /while \d+/)
        return_stmt =
find_return_in_holder(whiles[content[/\d+/].to_i], ifs, whiles)
        break if (return_stmt != nil)
      end
    end

    return return_stmt
  end

  def get_first_return_var(ifs, whiles)
    # Returns the variable name of the first return statement.
    # If the return statement doesn't contain a variable the
    # function returns "nil".
    return_stmt = find_return_in_holder(self, ifs, whiles)

    return "" if (return_stmt == nil)
    return_stmt = return_stmt.sub("return ", "")
    return return_stmt.strip
  end

  def general_name
    # Returns the general name of the function.
    # Replaces all "$<number>" with just "$"
    # Example: multiply_$1_$2 => multiply_$_$
    name = @name
    (1..@parameters.length).each do |i|
      name = name.sub("$#{i}", "$")
    end
    return name
  end
end

class IfHolder
  # Holder for all if statements.
  # "content" contains the content of the statement,
  # "logicstmt" contains the logical expression and
  # "othercont" contains the "otherwise" content
  attr_accessor :content, :logicstmt, :othercont
  def initialize(cont=[], logicstmt=nil, othercont=[])
    @content = cont
    @logicstmt = logicstmt
    @othercont = othercont
  end

  def equals?(ifholder)
    # Returns true if "ifholder" contains the same data as "self"
    return (ifholder.content.eql?(@content) && ifholder.logicstmt
== @logicstmt && ifholder.othercont.eql?(@othercont))
  end
end

class WhileHolder
  # Holder for all while statements
  # "content" contains the content of the statement and
  # "logicstmt" contains the logical expression
  attr_accessor :content, :logicstmt
  def initialize(cont=[], logicstmt=nil)
    @content = cont
    @logicstmt = logicstmt
  end

  def equals?(whileholder)
    # Returns true if "ifholder" contains the same data as "self"
    return (whileholder.content.eql?(@content) &&
whileholder.logicstmt == @logicstmt)
  end
end
```

5.3 __DEFAULTTHINGS__.rb

```
define $1_^_$2 as function with parameters(define a as Integer,
define b as Integer) {
   return ('a' ** 'b');
}

define $1_mod_$2 as function with parameters(define a as Integer,
define b as Integer) {
   return ('a' % 'b');
}

define puts as function {
   print "\n";
}

define Integer as class {
   define self as Integer;

   define initialize_$n as function {
      :self: = 0;
   }

   define set_$n_to_$1 as function with parameters(define a as
Integer) {
     :self: = 'a';
   }

   // PLUS //
   define $n_+_$1 as function with parameters(define a as Integer)
{
      return ('self' + 'a');
   }

   define $n_plus_$1 as function with parameters(define a as
Integer) {
      return ('self' + 'a');
   }

   define $n_+=_$1 as function with parameters(define a as Integer)
{
      :self: = ('self' + 'a');
   }

   // MINUS //
   define $n_-_$1 as function with parameters(define a as Integer)
{
      return ('self' - 'a');
   }

   define $n_minus_$1 as function with parameters(define a as
Integer) {
      return ('self' - 'a');
   }

   define $n_-=_$1 as function with parameters(define a as Integer)
{
      :self: = ('self' - 'a');
   }

   // MULTIPLIED //
   define $n_*_$1 as function with parameters(define a as Integer)
{
      return ('self' * 'a');
   }

   define $n_multiplied_by_$1 as function with parameters(define a
as Integer) {
      return ('self' * 'a');
   }

   define $n_*=_$1 as function with parameters(define a as Integer)
{
      :self: = ('self' * 'a');
   }

   // DIVIDED //
   define $n_/_$1 as function with parameters(define a as Integer)
{
      return ('self' / 'a');
   }

   define $n_divided_by_$1 as function with parameters(define a as
Integer) {
      return ('self' / 'a');
   }

   define $n_/=_$1 as function with parameters(define a as Integer)
{
      :self: = 'self' / 'a';
   }

   // MODULUS //
   define $n_%_$1 as function with parameters(define a as Integer)
{
      return ('self' % 'a');
   }

   define $n_mod_$1 as function with parameters(define a as
Integer) {
      return ('self' % 'a');
   }

   define $n_%=_$1 as function with parameters(define a as Integer)
{
      :self: = 'self' % 'a';
   }
   // POWER OF //
   define $n_**_$1 as function with parameters(define a as Integer)
{
      return ('self' ** 'a');
   }

   define $n_^_$1 as function with parameters(define a as Integer)
{
      return ('self' ** 'a');
   }

   define $n_to_the_power_of_$1 as function with parameters(define
a as Integer) {
      return ('self' ** 'a');
   }

   define $n_**=_$1 as function with parameters(define a as
Integer) {
      :self: = 'self' ** 'a';
   }

   define $n_^=_$1 as function with parameters(define a as Integer)
{
      :self: = 'self' ** 'a';
   }

   // INCREMENT //
   define increment_$n as function {
      :self: = ('self' + 1);
   }

   define increment_$n_by_$1 as function with parameters(define a
as Integer) {
      :self: = ('self' + 'a');
   }

   // DECREMENT //
   define decrement_$n as function {
      :self: = ('self' - 1);
   }

   define decrement_$n_by_$1 as function with parameters(define a
as Integer) {
      :self: = ('self' - 'a');
   }

   // EQUALITY //
   define $n_is_less_than_$1 as function with parameters(define a
as Integer) {
      return ('self' < 'a');
   }

   define $n_is_less_than_or_equal_to_$1 as function with
parameters(define a as Integer) {
      return :self: <= 'a';
   }

   define $n_is_greater_than_$1 as function with parameters(define
a as Integer) {
      return ('self' > 'a');
   }

   define $n_is_greater_than_or_equal_to_$1 as function with
parameters(define a as Integer) {
      return :self: >= 'a';
   }

   define $n_>=_$1 as function with parameters(define a as Integer)
{
     if ('self' == 'a') { return true; }
     if ('self' > 'a') { return true; }
     return false;
   }

   define $n_<=_$1 as function with parameters(define a as Integer)
{
     if ('self' == 'a') { return true; }
     if ('self' < 'a') { return true; }
     return false;
   }

   define $n_==_$1 as function with parameters(define a as Integer)
```

```
{
      return ('self' == 'a');
   }

   define $n_equals_$1 as function with parameters(define a as
Integer) {
      return ('self' == 'a');
   }

   define $n_does_not_equal_$1 as function with parameters(define a
as Integer) {
      define b as Boolean;
      :b: = :self: equals 'a';
      return !'b';
   }

   // PUTS //
   define puts_$n as function {
     print 'self';
     puts;
   }
}

define String as class {
   define self as String;

   define initialize_$n as function {
      :self: = "";
   }

   define set_$n_to_$1 as function with parameters(define a as
String) {
      :self: = 'a';
   }

   // EQUALITY //
   define $n_==_$1 as function with parameters(define a as String)
{
      return ('self' == 'a');
   }
   define $n_equals_$1 as function with parameters(define a as
String) {
      return ('self' == 'a');
   }

   define $n_does_not_equal_$1 as function with parameters(define a
as String) {
      define b as Boolean;
      :b: = :self: == 'a';
      return !'b';
   }

   // PUTS //
   define puts_$n as function {
      print 'self';
      puts;
   }
}

define Boolean as class {
   define self as Boolean;

   define initialize_$n as function {
      :self: = false;
   }

   define set_$n_to_$1 as function with parameters(define a as
Boolean) {
      :self: = 'a';
   }

   // EQUALITY //
   define $n_==_$1 as function with parameters(define a as Boolean)
{
      return ('self' == 'a');
   }

   define $n_equals_$1 as function with parameters(define a as
Boolean) {
      return ('self' == 'a');
   }
   define $n_does_not_equal_$1 as function with parameters(define a
as Boolean) {
      define b as Boolean;
      :b: = :self: == 'a';
      return !'b';
   }

   // AND //
   define $n_&&_$1 as function with parameters(define a as Boolean)
{
      return ('self' && 'a');
   }

   define $n_and_$1 as function with parameters(define a as
Boolean) {
      return ('self' && 'a');
   }

   // OR //
   define $n_||_$1 as function with parameters(define a as Boolean)
{
      return ('self' || 'a');
   }

   define $n_or_$1 as function with parameters(define a as Boolean)
{
      return ('self' || 'a');
   }

   // NOT //
   define not_$n as function {
      return !'self';
   }

   // PUTS //
   define puts_$n as function {
      print 'self';
      puts;
   }
}
```

```
define Float as class {
   define self as Float;

   define initialize_$n as function {
      :self: = 0.0;
   }

   define set_$n_to_$1 as function with parameters(define a as
Float) {
      :self: = 'a';
   }

   // PLUS //
   define $n_+_$1 as function with parameters(define a as Float) {
      return ('self' + 'a');
   }

   define $n_plus_$1 as function with parameters(define a as Float)
{
      return ('self' + 'a');
   }

   define $n_+=_$1 as function with parameters(define a as Float) {
      :self: = 'self' + 'a';
   }

   // MINUS //
   define $n_-_$1 as function with parameters(define a as Float) {
      return ('self' - 'a');
   }

   define $n_minus_$1 as function with parameters(define a as
Float) {
      return ('self' - 'a');
   }

   define $n_-=_$1 as function with parameters(define a as Float) {
      :self: = 'self' - 'a';
   }
   // MULTIPLIED //
   define $n_*_$1 as function with parameters(define a as Float) {
      return ('self' * 'a');
   }

   define $n_multiplied_by_$1 as function with parameters(define a
as Float) {
      return ('self' * 'a');
   }

   define $n_*=_$1 as function with parameters(define a as Float) {
      :self: = 'self' * 'a';
   }

   // DIVIDED //
   define $n_/_$1 as function with parameters(define a as Float) {
      return ('self' / 'a');
   }

   define $n_divided_by_$1 as function with parameters(define a as
Float) {
      return ('self' / 'a');
   }

   define $n_/=_$1 as function with parameters(define a as Float) {
      :self: = 'self' / 'a';
   }

   // MODULUS //
   define $n_%_$1 as function with parameters(define a as Float) {
      return ('self' % 'a');
   }

   define $n_mod_$1 as function with parameters(define a as Float)
{
      return ('self' % 'a');
   }

   define $n_%=_$1 as function with parameters(define a as Float) {
      :self: = 'self' % 'a';
   }
```

```
   // POWER OF //
   define $n_**_$1 as function with parameters(define a as Float) {
      return ('self' ** 'a');
   }

   define $n_^_$1 as function with parameters(define a as Float) {
      return ('self' ** 'a');
   }

   define $n_to_the_power_of_$1 as function with parameters(define
a as Float) {
      return ('self' ** 'a');
   }

   define $n_**=_$1 as function with parameters(define a as Float)
{
      :self: = 'self' ** 'a';
   }

   define $n_^=_$1 as function with parameters(define a as Float) {
      :self: = 'self' ** 'a';
   }

   // INCREMENT //
   define increment_$n as function {
      :self: = 'self' + 1;
   }

   define increment_$n_by_$1 as function with parameters(define a
as Float) {
      :self: = 'self' + 'a';
   }

   // DECREMENT //
   define decrement_$n as function {
      :self: = 'self' - 1;
   }

   define decrement_$n_by_$1 as function with parameters(define a
as Float) {
      :self: = 'self' - 'a';
   }

   // EQUALITY //
   define $n_==_$1 as function with parameters(define a as Float) {
      return ('self' == 'a');
   }

   define $n_equals_$1 as function with parameters(define a as
Float) {
      return ('self' == 'a');
   }

   define $n_>_$1 as function with parameters(define a as Float) {
      return ('self' > 'a');
   }

   define $n_is_greater_than_$1 as function with parameters(define
a as Float) {
      return ('self' > 'a');
   }

   define $n_>=_$1 as function with parameters(define a as Float) {
      if ('self' == 'a') { return true; }
      if ('self' > 'a') { return true; }
      return false;
   }

   define $n_is_greater_than_or_equal_to_$1 as function with
parameters(define a as Float) {
      if ('self' == 'a') { return true; }
      if ('self' > 'a') { return true; }
      return false;
   }

   define $n_<_$1 as function with parameters(define a as Float) {
      return ('self' < 'a');
   }

   define $n_is_less_than_$1 as function with parameters(define a
as Float) {
      return ('self' < 'a');
   }

   define $n_<=_$1 as function with parameters(define a as Float) {
      if ('self' == 'a') { return true; }
      if ('self' < 'a') { return true; }
      return false;
   }

   define $n_is_less_than_or_equal_to_$1 as function with
parameters(define a as Float) {
      if ('self' == 'a') { return true; }
      if ('self' < 'a') { return true; }
      return false;
   }

   define $n_does_not_equal_$1 as function with parameters(define a
as Float) {
      define b as Boolean;
      :b: = :self: equals 'a';
      return !'b';
   }

   // PUTS //
   define puts_$n as function {
     print 'self';
     puts;
   }
}

define Array as class {
   define self as Array;
   define length as Integer;

   define initialize_$n as function {
      :self: = [];
   }

   define set_$n_to_$1 as function with parameters(define a as
Array) {
      :self: = 'a';
      :length: = get length of :a:;
   }

   define get_length_of_$n as function {
      return 'length';
   }

   // EQUALITY //
   define $n_==_$1 as function with parameters(define a as Array) {
      return ('self' == 'a');
   }

   define $n_equals_$1 as function with parameters(define a as
Array) {
      return ('self' == 'a');
   }

   define $n_does_not_equal_$1 as function with parameters(define a
as Array) {
      define b as Boolean;
      :b: = :self: == 'a';
      return !'b';
   }

   // APPENDING //
   define add_Array_$1_to_$n as function with parameters(define a
as Array) {
      :self:<<'a';
      :length: += 1;
   }

   define add_String_$1_to_$n as function with parameters(define a
as String) {
      :self:<<'a';
      :length: += 1;
   }

   define add_Integer_$1_to_$n as function with parameters(define a
as Integer) {
      :self:<<'a';
      :length: += 1;
   }

   define add_Boolean_$1_to_$n as function with parameters(define a
as Boolean) {
      :self:<<'a';
      :length: += 1;
   }

   define add_Float_$1_to_$n as function with parameters(define a
as Float) {
      :self:<<'a';
      :length: += 1;
   }

   define add_index_$1_from_$2_to_$n as function with
parameters(define a as Integer, define b as Array) {
      :self:<<:b:['a'];
      :length: += 1;
   }

   // GETTING //
   define get_index_$1_from_$n as function with parameters(define
index as Integer) {
      return :self:['index'];
   }

   // PUTS //
   define puts_$n as function {
      print 'self';
      print "\n";
   }
}
```