LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Timescript

Författare

Albin Alvarsson, albal273@student.liu.se
Ludwig Holmberg, ludho208@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2020
Version 1.1

2020-05-25

Innehåll

1 Inledning

Vi på IP-programmet på Linköpings Universitet har under andra terminen av det första året av utbildningen arbetat med projektkursen TDP019: Konstruktion av datorspråk. Vi har i kursen bland annat arbetat med parsers och abstrakta syntaxträd vilket har byggt upp vår förståelse för vad som händer bakom kulisserna av ett programmeringsspråk för att sedan skapa vårt egna språk Timescript. I vår grupp valde vi att skapa ett indenteringskänsligt språk med den minsta möjliga mängden syntaktiskt socker som på ett enkelt vis skulle kunde behandla tid och klockslag direkt inbyggt i språkets vanliga satser och loopar. Ett språk där man kan specificera att en iteration ska köras mellan två klockslag eller under en vis tid med möjligheten att även bestämma hur ofta varje iteration får genomföras, samt möjligheten att skriva en if-sats med ett klockslag som krav.

2 Användarhandledning

Timescript är ett indenteringskänsligt scriptspråk som gör det enkelt för programmerare att hantera exekveringstider i olika kontrollstrukturer och loopar. Språket drar mycket inspiration från språk som Python och Ruby som är lätta syntaktiskta. Timescript är däremot inte objektorienterat och förväntas inte användas som språk i större arbeten där objektorientering kan vara att föredra.

Språket ses som ett skriptspråk som, utöver generella språks standardfunktioner, även kan användas för att skapa program som kör under specifika klockslag, under en viss tid framåt eller vid ett specifikt klockslag.

2.1 Installation

Innan språket kan användas behöver Ruby och Timescript installeras på systemet. För information om installation av Ruby se kapitel 2.1.2. För att installera Timescript börja med att ladda ner filen 'Timescript.zip'. Packa sedan upp filen på lämplig plats på datorn. Med nuvarande installation av Timescript krävs det att filen som ska köras ligger i Timescripts installationsfolder. För information om en systemomfattande installation se kapitel 2.1.3 för Linux och Mac OS eller kapitel 2.1.4 för Windows. För att köra Timescript-kod öppna en terminal, navigera till Timescript installationsfolder och skriv kommandot:

```
~$ ruby timescript.rb [Filename]
```

Utesluts [Filename] startar Timescript i interpreterat läge. För mer information om interpreterat läge se kapitel 2.2.

2.1.1 Allmänna krav

Här följer allmänna krav som krävs för arbete och körning av Timescript-kod.

- Ruby v2.5.1 eller senare (språket är skrivet i 2.5.1 men förväntas fungera bakåt till Ruby v1.8.7).
- En text-editor för att skriva Timescript-kod. (Behövs ej om användaren inte har som avsikt att skriva egen kod.)

2.1.2 Installation av Ruby

För att använda Timescript och köra programkod skrivet med språket behöver Ruby 1.8.* eller senare finnas installerat på datorn. För att se om Ruby är installerat kan du på ett Linux, Mac OS eller annat unixbaserat system öppna en terminal med tangenterna alt + t och sedan i terminalen skriva den översta raden av nedanstående block:

```
~$ ruby --version
   ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux-gnu]
```

Blir den egna utskriften inte som den ovan finns inte Ruby installerat på systemet och behöver installeras. Det kan göras med följande kommandon:

```
~$ sudo apt update
~$ sudo apt install ruby-full
```

Med kommandot *sudo apt update* blir datorns interna lista på nedladdningsbara program uppdaterad med eventuellt ny information om version och adress för nedladdning. Kommandot *sudo apt install ruby-full* installerar sedan Ruby. Läs utskrifterna under installationen noga eftersom installationen kräver korrekt inskriven indata för att gå igenom. För att verifiera att installationen genomförs utan problem, skriv åter kommandot *ruby –version*. Blir utskriften fortfarande inte korrekt blev något med stor sannolikhet fel under installationen och vi uppmanar er därför att åter skriva in kommandona i blocket ovan och ta extra hänsyn till vad som skrivs ut för att veta vad installationen kräver för indata för att genomföras utan problem.

Ska Timescript användas på ett Windows-system kan Ruby istället installeras genom att hämta hem språket från Rubys webbsida https://www.ruby-lang.org/en/ och efter det starta den nedladdade filen med ett klick. Därefter följ instruktionerna i installationsprogrammet.

2.1.3 Timescript på Linux och Mac OS

För att göra Timescript lättare att använda på Linux och Mac OS gör som följer:

1. Öppna 'timescript.rb' med valfri editor och skriv på rad två mellan de två enkla citattecknen in den absoluta installationsvägen för språket. Den absoluta installationsvägen för språket kan fås genom att navigera till Timescripts installationmapp med terminalen och skriva in kommandot *pwd* som skriver ut den absoluta vägen till nuvarande öppen folder.
2. Skriv kommandot *chmod a+x timescript.rb* vilket gör timescript-filen exekverbar. För att genomföra kommandot behöver terminalen arbeta i samma folder som timescript.rb.
3. Navigera terminalen till systemets root folder och öppna filen .bash_aliases. Om filen inte finns så behöver du skapa den. I filen skriv *alias Timescript='<Install_path>/timescript.rb'* där <Install_path>är samma installationsväg som den från steg 1.

Timescript-program kan nu lagras var som helst på systemet och köras oavsett var på datorn terminalen arbetar med kommandot:

```
~$ Timescript [Filename]
```

2.1.4 Timescript på Windows

För att göra Timescript lättare att använda på ett Windows-system gör som följer:

1. Öppna 'timescript.rb' med valfri editor och skriv på rad två mellan de två enkla citattecknen in den absoluta installationsvägen för språket. Lägg till ett extra snedstreck bakom varje snedstreck i installationsvägen och även två snedstreck sist i strängen (Ex. 'C:\\Timescript\\').
2. Lägg till installationsvägen från föregående steg (utan dubbelt snedstreck) i Windows miljövariabler 'Path'. Navigera till Windows miljövariabler genom att klicka på Windows 'Startmenyn' och skriv 'Kontrollpanel'. Navigera därefter till 'System och Säkerhet' ->'System' ->'Avancerade Systeminställningar' ->'Miljövariabler'. Dubbelklicka på 'Path' i boxen 'Systemvariabler'. Klicka 'New' och lägg till installationsvägen från föregående steg (utan dubbelt snedstreck, Ex. 'C:\Timescript\'). Klicka OK.

Timescript-program kan nu lagras var som helst på systemet och köras oavsett var på datorn terminalen arbetar med kommandot:

```
~$ Timescript [Filename]
```

2.2 Interpreterat läge

Startas Timescript utan att specificera en programfil startar språket i interpreterat läge. I den interpreterade miljön kan användaren skriva in enklare operationer som matematiska beräkningar och variabeltilldelning. Det interpreterade läget klarar inte av konstruktioner som sträcker sig över flera rader som if-sater och loopar. För att avsluta skriv antigen 'quit', 'exit' eller en tom rad. Inget av det som skrivs i den interpreterade miljön sparas efter nedstängning.

2.3 Reserverade nyckelord

I Timescript finns flera reserverade nyckelord, ord som är reserverade för språkets olika konstruktioner och som därför inte får användas som variabel- eller funktionsnamn. Dessa är:

true	else	for	nil	and
false	break	while	0DQdhzc2x8	not
if	return	from	input	
elsif	def	at	or	

2.4 Datatyper

I Timescript kan du använda dig av datatyperna Integer, Float, Boolean, String och List. Det finns även en datatyp *nil* som representerar ett icke existerande värde.

2.4.1 Integer

Datatypen Integer omfattar alla positiva och negativa heltal.

Exempel – Syntax

```
1.  5
2.  -5
```

2.4.2 Float

Datatypen Float omfattar alla positiva och negativa decimaltal.

Exempel – Syntax

```
1.  5.0
2.  -5.0
```

2.4.3 Boolean

Datatypen Boolean omfattar sanningsvärdena *true* och *false*.

Exempel – Syntax

```
1.  true
2.  false
```

2.4.4 String

Datatypen String omfattar en teckenföljd.

Exempel – Syntax

```
1.  "String"
2.  'This is a string'
```

2.4.5 List

Datatypen List omfattar en samling av element som kan bestå av datatyperna *Integer*, *Float*, *Boolean*, *String* och *List*.

Exempel – Syntax

```
1.  [1, 2, 4, 8]

    En lista som innehåller elementen 1, 2, 3, 4 och 8.

2.  [1, 2.0, true, "4", [8]]

    En lista som innehåller elementen 1, 2.0, true, "4" och en lista med elementet 8.
```

Exempel – Variabeltilldelning och uttag av data

```
1.  var = [1, 2, 4]          => [1, 2, 4]
2.  var[0]                   => 1

    Tar ut värdet på index 0 av listan 'var' som är 1.

3.  var[2]                   => 4

    Tar ut värdet på index 2 av listan 'var' som är 4.
```

Exempel – Tillägg av data i befintlig lista

```
1.  var = [1, 2, 4]          => [1, 2, 4]
2.  var << 8                 => [1, 2, 4, 8]

    Lägger till värdet 8 sist i listan 'var'.
```

Exempel – Ta bort data ur lista

```
1.  var = [1, 2, 4]          => [1, 2, 4]
2.  delete_at(var, 1)        => [1, 4]

    Tar bort elementet på index 1 ur listan 'var', dvs tar bort det andra elementet ur
    listan.
```

2.5 Variabler och tilldelning

En variabel är en namngiven behållare som kan spara undan data och information för senare användning i ett program. Timescript använder sig av dynamisk typning vilket innebär att användaren av språket inte behöver specificera en datatyp för varje variabel utan det sker automatiskt bakom kulisserna vid tilldelning. Variabeltilldelning sker med likhetstecknet (=).

Exempel – Tilldelning

```
1.  var = 5                 => Integer
2.  var = 5.0               => Float
3.  var = true              => Boolean
4.  var = "hello"           => String
5.  var = [1, 2, 3]         => List
```

Det går även att använda aritmetiska operatorer med likhetstecknet för att utföra en beräkning på en variabel och tilldela resultatet av beräkningen till variabeln.

Exempel – Tilldelning med aritmetisk operator

```
    var = 10                => 10
1.  var += 2                => 12
2.  var -= 2                => 8
3.  var *= 2                => 20
4.  var /= 2                => 5
```

2.6 Operatorer

En operator använder ett eller flera värden för att göra en beräkning.

2.6.1 Aritmetiska operatorer

Nedan följer språkets aritmetiska operatorer.

```
Namn:               Symbol:     Utfall:

Addition:           +           10 + 2   => 12
Subtraktion:        -           10 - 2   => 8
Multiplikation:     *           10 * 2   => 20
Division:           /           10 / 2   => 5
Potens:             **          10 ** 2  => 100
```

2.6.2 Logiska operatorer

Nedan följer språkets logiska operatorer.

```
Namn:               Symbol:     Utfall:

Konjunktion:        and         true and true    => true
                                true and false   => false
                                false and false  => false

Disjunktion:        or          true or true     => true
                                true or false    => true
                                false or false   => false

Negation:           not         not true         => false
                                not false        => true
```

2.6.3 Jämförelseoperatorer

Nedan följer språkets jämförelseoperatorer.

```
Namn:                           Symbol:         Utfall:

Mindre än:                      <               1 < 2     => true
Större än:                      >               1 > 2     => false
Ekvivalent:                     ==              1 == 2    => false
Mindre än eller ekvivalent:     <=              1 <= 2    => true
Större än eller ekvivalent:     >=              1 >= 2    => false
Ej ekvivalent:                  !=              1 != 2    => true
```

2.7 Inbyggda funktioner

Inbyggda funktioner är funktioner som är fördefinierade av Timescript.

```
Namn:                   Parametrar:                     Beskrivning:

print(x)                x: String, Integer, Float,      Tar en parameter och skriver ut
                           Boolean, List                det inskickade argumentet till
                                                        konsolen.
                                                        Skickas ingen parameter skrives
                                                        det istället ut ett radbryt.

len(x)                  x: List                         Returnerar längden på den
                                                        inskickade listan.

int(x)                  x: String, List                 Försöker konvertera det inskickade
                                                        värdet till en Integer.
                                                        Raise ... om konverteringen
                                                        misslyckas.

str(x)                  x: Integer, Float               Konverterar det inskickade värdet
                                                        till en String.

remove_at(x, y)         x: List                         Tar bort ett element ur List på
                        y: Integer                      index Integer och returnerar den
                                                        uppdaterade listan.
```

2.8 Indata

Med nyckelordet ’input’ kan användare ge programmet egen indata under körning.

Exempel – Input

```
data = input

# Vi ger indatan "Hello!"

=> Hello!
```

2.9 Kontrollsatser

För att kontrollera vilken kod i en sats som ska exekveras används kontrollsatser. Följande kontrollsatser är tillgängliga i Timescript.

2.9.1 If-satser

En If-sats används för att endast exekvera ett kodblock om ett villkor är sant. Om villkoret evalueras till falskt så kommer koden innanför If-statsen inte att exekveras.

Exempel – Grundläggande If-sats

```
if true
    print("Hello!")

=> Hello!
```

```
if false
    print("Hello!")

=>
```

Ett villkor kan bestå av en jämförelse med en variabel.

Exempel – If-sats med variabel

```
x = 5
if x < 10
    print("Hello!")

=> Hello!
```

Det går även att kombinera flera villkor med hjälp av logiska operatorer.

Exempel – If-sats med flera villkor

```
x = 5
if (x < 10) and (2 == 2)
    print("Hello!")

=> Hello!
```

Elsif-satsen används för att utvärdera ett nytt villkor om tidigare villkor inte varit sanna.

Exempel – Grundläggande Elsif-sats

```
if false
    print("Hello!")
elsif true
    print("World!")

=> World!
```

Det går att använda flera Elsif-satser efter varandra för att utvärdera flera villkor.

Exempel – If-sats med flera Elsif-satser

```
if false
    print("Hello!")
elsif false
    print("World!")
elsif true
    print("Here!")

=> Here!
```

Else-satsens kodblock exekveras endast om inget av tidigare villkor har varit sanna.

Exempel – Grundläggande Else-sats

```
x = 5
if x == 3
    print("Hello!")
elsif x == 4
    print("World!")
else
    print("Here!")

=> Here!
```

2.9.2 At-satser

At-satsen används för att exekvera ett kodblock vid ett satt klockslag. Klockslaget anges i 24-timmarsformat. När exekveringen når en at-sats stannar programmet fram till att systemets interna klocka blir det satta klockslaget innan koden i satsen exekveras. Ligger at-satsen i en while-loop kommer kodblocket exekveras en gång vid det angivna klockslaget varje dag fram till att while-loopen eller programmet avslutas. För mer information om while-loopar se kapitel 2.10.2

Exempel – Grundläggande At-sats

```
at 14:30:00
    print("Hello!")

=> Hello!
```

2.10 Iteration

Iterationssatser används för att upprepa ett kodblock flera gånger tills ett villkor är uppfyllt. När det satta villkoret är inte längre är sant så avslutas iterationen och programmet fortsätter att utvärdera koden efter loopen.

2.10.1 For-loopar

For-loopen itererar igenom ett kodblock ett visst antal gånger som är fördefinierat av användaren.

Exempel – Grundläggande For-loop

```
for i = 0, i < 2, i += 1
    print("Hello!")

=> Hello!
=> Hello!
=> Hello!
```

För att avbryta en iteration innan villkoret evalueras till falskt så går det att avbryta med *break*. Då avslutas iterationen omedelbart och programmet fortsätter att utvärdera koden efter loopen.

Exempel – For-loop med break

```
for i = 0, i < 2, i += 1
    print("Hello!")
    if (i == 1)
        break

=> Hello!
=> Hello!
```

For-loopens villkor kan bestå av ett tidsintervall som anges i timmar, minuter och sekunder. For-loopen kommer då att iterera igenom kodblocken under det tidsintervall som anges.

Exempel – For-loop med tidsintervall

```
# Följande kod kommer att exekveras i 8 timmar, 45 minuter och 30 sekunder.

for 8h 45m 30s
    print("Hello!")

=> Hello!
=> Hello!
...
```

För att iterera inom ett tidsintervall behöver inte användaren ange alla tre tidsenheter vilket innebär att användaren själv kan välja att ange tiden på det sätt som känns lämpligast.

Exempel – For-loop med alternativa tidsintervall

```
# Följande kod kommer att exekveras i 100 minuter (1 timme och 40 minuter).

for 100m
    print("Hello!")

=> Hello!
=> Hello!
...
```

```
# Följande kod kommer att exekveras i 300 sekunder (5 minuter).

for 300s
    print("Hello!")

=> Hello!
=> Hello!
...
```

each används för att begränsa antalet gånger som loopen itererar igenom kodblocket under tidsintervallet. Detta intervall definieras på samma sätt som for-loopens tidsintervall.

Exempel – For-loop med tidsintervall och each

```
# Följande kod kommer att exekveras i 30 sekunder var femte sekund.

for 30s each 5s
    print("Hello!")

=> Hello!
=> Hello!
...
```

2.10.2 While-loopar

While-loopen itererar igenom ett kodblock så länge ett satt villkor är sant.

Exempel – Grundläggande While-loop

```
x = 0
while x < 2
    print("Hello!")
    x += 1

=> Hello!
=> Hello!
=> Hello!
```

För att avbryta en iteration innan villkoret evalueras till falskt så går det att avbryta med *break*. Då avslutas iterationen omedelbart och programmet fortsätter att utvärdera koden efter loopen.

Exempel – While-loop med break

```
x = 0
while x < 2
    print("Hello!")
    if (x == 1)
        break
    x += 1

=> Hello!
=> Hello!
```

each används för att begränsa hur ofta loopen itererar igenom kodblocket. Detta intervall definieras genom att ange antalet timmar, minuter och sekunder (_*h* _*m* _*s*).

Exempel – While-loop med each

```
# Följande kod kommer att exekveras var femte sekund.

x = 0
while x < 2 each 5s
    print("Hello!")
    x += 1

=> Hello!
=> Hello!
=> Hello!
```

2.10.3 From-loopar

From-loopen itererar igenom ett kodblock under ett tidsintervall mellan två klockslag. Klockslagen anges i 24-timmarsformat. Kodblocket kommer att först exekveras vid det först angivna klockslaget och därefter iterera till det andra angivna klockslaget.

Exempel – Grundläggande From-loop

```
# Följande kod kommer att exekveras från kl. 14:00:00 till 15:00:00.

from 14:00:00 to 15:00:00
    print("Hello!")

=> Hello!
=> Hello!
...
```

each används för att begränsa hur ofta loopen itererar igenom kodblocket. Detta intervall definieras genom att ange antalet timmar, minuter och sekunder (*_h _m _s*).

Exempel – From-loop med each

```
# Följande kod kommer att exekveras från kl. 14:00:00 till 15:00:00 var tionde minut.

from 14:00:00 to 15:00:00 each 10m
    print("Hello!")

=> Hello!
=> Hello!
...
```

2.11 Funktioner

En funktion är ett kodblock som kan anropas oavsett var i koden exekveringen är. Funktioner gör det möjligt att återanvända ofta använda kodstycken utan att behöva skriva om kodstycket varje gång det ska användas. En funktion måste vara deklarerad innan den kan användas.

Exempel – Deklarering av en funktion

```
def test_func()
    print("This is")
test_func()
print("a test!")

=> This is
=> a test!

    När funktionen anropas hoppar programmet till kodblocket definierad i funktionen och
    skriver ut "This is" innan den går tillbaka till funktionskallelsen och fortsätter
    nedåt i koden och skriver ut "a test!".
```

2.11.1 Parametrar

Beroende på syfte kan det finnas anledning att skicka in data och variabler i en funktion, något som är möjligt med parametrar.

Exempel – Funktion med parametrar

```
def param_func(a, b)
    print(a + b)
param_func(5, 10)

=> 15

    När funktionen anropas skickas värdet 5 till parameter a medan värdet 10 skickas till
    parameter b. I funktionen param_func finns vid denna kallelse alltså variabeln a med vä
    rdet 5 och variabeln b med värdet 10.
```

2.11.2 Returnerande funktioner

En funktion kan utöver att ta emot data i form av parametrar skicka tillbaka data till ett funktionsanrop med nyckelordet 'return'. När exekveringen av en funktions kodblock når en 'return' avbryts all exekvering i funktionen och uttrycket bredvid returneras till funktionsanropet.

Exempel – Funktion med return

```
def return_func(a, b)
    if a + b < 20
        return "a + b is less than 20!"
    else
        return "a + b is equal to or greater than 20!"
print(param_func(9, 9))

=> "a + b is less than 20!"

    När funktionen anropas med värdena 9 och 9 är a + b = 18 vilket är mindre än 20.
    Strängen "a + b is less than 20!" returneras och skrivs ut.

print(param_func(10, 11))

=> "a + b is equal to or greater than 20!"

    När funktionen anropas med värdena 10 och 11 är a + b = 21 vilket är större än 20.
    Strängen "a + b is equal to or greater than 20!" returneras och skrivs ut.
```

2.11.3 Rekursivitet

När en funktion i sitt kodblock kallar på sig själv kallas funktionen för en rekursiv funktion. Det är viktigt att i en rekursiv funktion alltid ha en kontroll-sats som stannar rekursionen när den har nått sitt mål så att programmet inte fastnar i en oändlig loop av funktionsanrop.

Exempel – Rekursiv funktion

```
def rec_func(i)
    if i < 10
        i += 1
        return rec_func(i)
    else
        return i
print(rec_func(0))

=> 10

    Funktionen rec_func kallar på sig själv 10 gånger innan basfallet är uppfyllt och
    processen att returnera 'i' börjar.
```

2.12 Indentering och Scope

Timescript är ett indenteringskänsligt språk vilket innebär att det är viktigt att hålla reda på nuvarande indenteringsnivå när kod skrivs. Indenteringsnivån är kopplad till nuvarande scope och bestämmer vilka variabler användaren och språket har tillgång till på olika rader i koden. När ett variabelscope skapas överförs alla variabler och sparad data från föregående scope till det nya. Kod med en högre indenteringsnivå har alltid tillgång till variabler med en lägre indenteringsnivå.

Exempel – Variabelscope

```
x = 10
if true
    print(x)

=> 10

    Eftersom det indenterade blocket i if-satsen har en högre indenteringsnivå än blocket
    utanför har blocket tillgång till variabeln 'x' från föregående scope vilket ger
    utskriften '10'.
```

Exempel – Variabelscope 2

```
if true
    x = 20
print(x)

=> #Programmet krashar

    Programmet krashar med felmeddelandet 'Variable x does not exist!' eftersom variabeln y
    skapades i ett block med en högre indenteringsnivå än det block där print kallas. När
    koden går ur det indenterade blocket i if-satsen raderas den scopenivån och variabeln
    'x' försvinner.
```

Exempel – Variabelscope 3

```
x = 10
if true
    x = 20
print(x)

=> 20

    Eftersom variabeln 'x' skapas innan if-satsen finns den tillgänglig i det indenterade
    blocket. När koden går ur blocket överförs värdet på föregående variabel 'x' till
    variabel 'x' på den nya indenteringsnivå.
```

2.12.1 Funktionsscope

Ett funktionsscope skiljer sig från normala scope eftersom de inte tar med sig alla variabler och sparad data från föregående scope när de skapas. De tar endast med sig de parametrar som skickas in i dem som argument.

Exempel – Funktionsscope

```
x = 10
def func(x)
    print(x)

=> 10

    Eftersom variabel 'x' skickas till funktionen func som en parameter har det indenterade
    blocket i funktionen tillgång till värdet på variabel 'x' vilket ger utskriften '10'.
```

Exempel – Funktionsscope 2

```
x = 10
def func()
    print(x)

=> #Programmet krashar

    Eftersom variabel 'x' inte skickas till funktionen func som en parameter har det
    indenterade blocket i funktionen inte tillgång till variabel 'x' vilket orsakar
    programmet att krasha med felmeddelandet 'Variable x does not exist!'.
```

3 Systemdokumentation

3.1 Språkbeskrivning

När ett program körs i Timescript kontrollerar språket att alla filer som behövs för att interpretera koden finns. Därefter läser timescript in den specificerade filen som en sträng som sedan delas upp i tokens av lexern. När all kod har delats upp i tokens tolkas de av parsern genom att matchas mot regler som bygger upp ett syntaxträd av noder. När hela filen är parsad och syntaxträdet är fullständigt evalueras alla noder i trädet och timescript koden börjar köras.

3.1.1 Lexer

Lexern får in en sträng bestående av all programkod. Med hjälp av reguljära uttryck matchar lexern alla tecken från strängen mot förbestämda mönster. Vid en matchning skickar lexern ut antingen en token som i exemplet nedan eller den sträng som matchade mönstret.

```
    token(/\d{2}:\d{2}:\d{2}/) {|m| Abs_time_token.new(m)}
```

Mönstret i exemplet ovan försöker matcha en del av programkoden mot ett klockslag (*Ex. 14:30:00*) för att sedan skapa en *Abs_time_token* med strängen som den matchade med. Denna token kan senare evalueras i syntaxträdet när hela strängen har tokeniserats.

3.1.2 Parser

När lexern har skapat tokens av hela programsträngen kommer alla tokens sedan att tolkas av parsen och matchas mot språkets grammatiska regler. När parsern hittar en matchning med en regel skapas en lämplig klass som sedan skickas vidare uppåt i syntaxträdet till syntaxträdets startnod där startnoden evalueras. En regel i parsern kan se ut på följande sätt:

```
    rule :integer do
        match('-', Integer) {|neg, a| Integer_node.new(a, neg)}
        match(Integer) {|a| Integer_node.new(a)}
    end
```

Om parsern matchar med regeln *:integer* skapas en *Integer_node* som senare kan evalueras och returnera sitt värde.

3.1.3 Noder

När det specificerade Timescript-programmet har lästs in fullständigt kommer det att finnas en instans av klassen *Program_node* som fungerar som en basnod, och det är denna nod som innehåller alla andra noder som skapats av parsern. En nodklass kan exempelvis se ut på följande sett:

```
class Addition_node
  def initialize(a, b)
    @a = a
    @b = b
  end

  def eval
    @a.eval + @b.eval
  end
end
```

När en nod skapas körs funktionen *initialize* som sätter klassens instansvariabler till de inskickade värdena. I fallet av en *Addition_node* är de två argumenten ofta *Integer_node*s eller *Float_node*s, som själva har varsin initialize- och eval-funktion.
I funktionen eval evalueras den kod som är unik för varje nod. I fallet med en addition_node kommer den i sin eval metod först kalla på instansvariablernas evalfunktioner och därefter returnera värdet av de två instansvariablerna adderade med varandra.

3.1.4 Syntaxträd

När alla noder är skapade kommer de börja bli evalueras utifrån basnoden *Program_node*. När basnodens eval funktion körs börjar basnoden kalla på sina instansvariablers eval-funktioner som i sin tur är noder som kallar på sina instansvariablers eval-funktioner. Processen går hela vägen ut i syntaxträdets alla grenar innan noderna börjar returnera resultatet av deras eval-funktioner tillbaka till basnoden. När eval-funktionerna har arbetat sig tillbaka till basnoden har hela programmet exekverats.

3.2 Scopehantering

Scope är i Timescript implementerat med hjälp av Scope_handler-klassen. Där lagras variabler från samma scope i unika Ruby hashes. Dessa hashes lagras sedan i en gemensam lista där den sista hashen är nuvarande scope. Även deklarerade funktioner lagras i Scope_handler-klassen men dessa lagras direkt i en gemensam hash.

För att hantera variabler och scope mellan olika kontroll- och iterationssatser används följande inbyggda funktioner.

- **add_variable_scope**: Lägger till ett nytt scope sist i scope-listan som innehåller alla variabler från föregående scope.
- **pop_variable_scope**: Tar bort det sista scopet i scope-listan. Om det nästkommande sista scopet innehåller variabler med samma namn som föregående sista scope överförs värdet på dessa. Det vill säga om en variabel 'number' finns i både sista och näst sista scopet överförs värdet på den 'number' som fanns i sista scopet till den i näst sista scopet.
- **assign_variable**: Lagrar variabelnamn och värdet på variabeln i den hash som är det nuvarande scopet.
- **get_variable**: Hämtar värdet på en variabel med det angivna namnet i nuvarande scope. Ger ett felmeddelande om variabeln inte existerar.

För att hantera variabler och scope mellan funktioner används följande inbyggda funktioner.

- **add_function_scope**: Lägger till ett nytt scope sist i scope-listan som innehåller alla parametrar från funktionskallet.
- **pop_function_scope**: Tar bort det sista scopet i scope-listan.
- **pop_till_scope**: Tar bort det sista scopet i scope-listan tills det bara finns ett scope kvar, eller tills nästkommande scope är av samma typ inparametern.
- **assign_func**: Lagrar funktionsnamn, parametrar och kodblock i funktions-hashen.
- **run_func**: Kör en lagrad funktion med det inskickade namnet och argument-listan. Run_func kollar om en funktion med det inskickade namnet existerar som en fördefinierad språkfunktion eller som en funktion användaren har skapat. Om ingen funktion med det inskickade namnet hittas ges ett felmeddelande, annars exekveras funktionens kodblock.

Utöver det innehåller Scope_handler klassen dessutom getters och setters för hanteringen av return och break.

3.2.1 Kodstandard

Vi har inte följt någon specifik kodstandard, men vi har alltid försökt att vara konsekventa i sättet att skriva vår kod. Vi har använt samma indentering på samtliga ställen i koden och våra variabelnamn följer samma stil. Detta stämmer även för funktions- och klassnamn.

3.2.2 BNF

```
program ::= statement_list

statement_list ::= statement_list line
                 | line

line ::= statement newline
       | statement

statement ::= return
            | break
            | function
            | at_stmt
            | while_loop
            | from_loop
            | for_loop
            | if_else_stmt
            | assignment
            | expr

param_list ::= param_list ',' String
             | String

arg_list ::= arg_list ',' expr
           | expr

function ::= 'def' String '(' param_list ')' indented_block
           | 'def' String '(' ')' indented_block

function_call ::= '/\w+/' '(' arg_list ')'
                | '/\w+/' '(' ')'

at_stmt ::= 'at' abs_time indented_block

from_loop ::= 'from' abs_time 'to' abs_time 'each' rel_time indented_block
            | 'from' abs_time 'to' abs_time indented_block

for_loop ::= 'for' rel_time 'each' rel_time indented_block
           | 'for' rel_time indented_block
           | 'for' assignment ',' comparison ',' assignment indented_block

rel_time ::= '/\d+h/' '/\d+m/' '/\d+s/'
           | '/\d+h/' '/\d+m/'
           | '/\d+h/' '/\d+s/'
           | '/\d+m/' '/\d+s/'
           | '/\d+h/'
           | '/\d+m/'
           | '/\d+s/'

abs_time ::= Abs_time_token

while_loop ::= 'while' boolean_expr 'each' rel_time indented_block
             | 'while' boolean_expr indented_block

if_else_stmt ::= if_stmt elsif_stmt else_stmt
               | if_stmt elsif_stmt
               | if_stmt else_stmt
               | if_stmt

if_stmt ::= 'if' boolean_expr indented_block

elsif_stmt ::= 'elsif' boolean_expr indented_block elsif_stmt
             | 'elsif' boolean_expr indented_block
```

```
else_stmt ::= 'else' indented_block

indented_block ::= newline indent statement_list newline dedent
                 | newline indent statement_list dedent
                 | newline indent statement_list

assignment ::= var '[' arithmetic_expr ']' '=' expr
             | var '=' expr
             | var shorthand_op expr

shorthand_op ::= '+='
               | '-='
               | '*='
               | '/='

expr ::= nil
       | input
       | list
       | list_shorthand_add
       | boolean_expr
       | String_token

break ::= 'break'

return ::= 'return' expr

nil ::= 'nil'

input ::= 'input'

list ::= '[' arg_list ']'
       | '[' ']'

list_shorthand_add ::= var '<<' expr

boolean_expr ::= boolean_expr 'or' boolean_term
               | boolean_term

boolean_term ::= boolean_term 'and' boolean_factor
               | boolean_factor

boolean_factor ::= 'not' boolean_factor
                 | 'true'
                 | 'false'
                 | comparison

comparison ::= arithmetic_expr comp_op arithmetic_expr
             | arithmetic_expr

comp_op ::= '<'
          | '<='
          | '>'
          | '>='
          | '=='
          | '!='

arithmetic_expr ::= arithmetic_expr '+' arithmetic_term
                  | arithmetic_expr '-' arithmetic_term
                  | arithmetic_term
```

```
arithmetic_term ::= arithmetic_term '*' arithmetic_factor
                  | arithmetic_term '/' arithmetic_factor
                  | arithmetic_factor

arithmetic_factor ::= arithmetic_primary '**' arithmetic_factor
                    | arithmetic_primary

arithmetic_primary ::= '(' boolean_expr ')'
                     | integer
                     | float
                     | list_var
                     | function_call
                     | var

list_var ::= list_var '[' arithmetic_expr ']'
           | var '[' arithmetic_expr ']'

var ::= String

integer ::= '-' Integer
          | Integer

float ::= '-' Float
        | Float
```

4 Erfarenheter och Reflektion

Under arbetet med Timescript tycker vi att vi har lyckats lägga upp arbetet bra. Efter att ha jobbat med projektet i ungefär en vecka gjorde vi en simpel tidsplanering för resten av kursen, en tidsplanering som visade sig spegla den verkliga tidsåtgången relativt bra. Under den första veckan av arbete med språket gick allt framåt mycket enkelt då det mesta av det som implementerades var sådant vi redan arbetat med och gått igenom under föreläsningar tidigare.

Vårt första problem fick vi när vi började implementera jämförelseoperatorer. Vår dåvarande lexer klarade inte av att matcha efter två på varandra följande tecken (tecken som =, +, <osv) vilket gjorde att våra parserregler för jämförelseoperatorer blev svåra att skriva på ett bra sätt. Oavsett hur vi skrev våra regler blev en parsning som '1 <= 2' accepterat och parsad korrekt medan en parsning av '1 <= 2' misslyckades då lexern inte kunde matcha någon sträng med '<='. Vi bokade in ett handledningstillfälle för att lösa problemet, vilket vi tack vare handledningen lyckades lösa, men blev istället medvetna om ett annat. I vår dåvarande kod parsade vi uttryck direkt när vi gjorde en matchning i våra regler istället för att skapa ett syntaxträd. Skulle vi ha behållit vår direkta parsning av uttryck istället för att byta mot det abstrakta syntaxträdet skulle programmets körtid ökat ordentligt. Implementationen av syntaxträdet gick oväntat enkelt och fort tack vare att vi än inte hade implementerat många funktioner i språket.

Redan innan vi började implementera språkets indenteringskänslighet hade vi aningar om att det var en sådan sak som kunde bli svårt och ta tid. Vi valde att ta upp frågan tidigt i arbetet och bokade ett handledningpass där vi fick ta del av ett kodstycke från vår handledares indeteringskänsliga språk. Till en början hade vi några problem att få allt att fungera som det skulle men när allt fungerade var det tydligt varför det fungerade och kunde lätt släppa det för att aldrig behöva tänka på indenteringen igen.

Scopehanteringen i vårt språk har under utvecklingen av språket varit under konstant ändring eftersom väldigt många av språkets delar kan påverka ett scope. När vi påbörjade utvecklingen av språket gick vi in blinda på hur scopet skulle fungera då vi inte hade någon kunskap eller erfarenhet av hur ett scope faktiskt fungerade. Detta ledde till konstanta ändringar i scopehanteringskoden vilket ledde till mycket extra utvecklingstid som hade kunnat spenderats på andra funktioner av språket. Vi är inte säkra på om vi skulle ha kunnat göra en bra plan för scope hantering innan vi började arbetet då vi som flera gånger tidigare nämnt inte hade någon tidigare kunskap i området och heller inte visste hur vi skulle införskaffa den.

Något som starkt underlättade vår utvecklingsprocess under arbetet var vår utförliga testfil där vi skapade tester med Rubys test/unit. I filen skrev vi tester för samtliga möjliga parsningar som skulle kunna tänkas ske under körning. Med filen upptäckte vi direkt om någon av våra föregående konstruktioner hade slutat fungera som följd av implementationen av en ny funktion i språket. Tack vare testerna hittade vi flera parsningsproblem vi troligen aldrig skulle ha upptäckt utan dem.

Allt som allt tycker vi arbetet med konstruktionen av det egna datorspråket har gått oväntat lätt framåt där det inte har varit något vi känt att vi verkligen har fastnat på. Vi lyckades följa vår ursprungliga plan med få avvikelser där arbetsinsatsen under arbetet till stora delar har varit jämn.

5 Programkod

5.1 timescript.rb

```
#!/usr/bin/env ruby
$install_directory = ''

require 'pathname'

if $install_directory != ''
  if Pathname($install_directory).exist?
    require $install_directory + 'rdparse.rb'
    require $install_directory + 'nodes.rb'
  else
    raise LoadError.new("Invalid install directory!")
  end
else
  if Pathname(Dir.pwd).each_child(false).any? {|f| f.basename.to_s == 'timescript.rb'}
    require_relative 'rdparse.rb'
    require_relative 'nodes.rb'
  else
    raise LoadError.new("No timescript files found in this folder!")
  end
end

class String_token
  attr_accessor :string
  def initialize(a)
    @string = a
    @string[0] = ''
    @string[@string.length - 1] = ''
  end
end

class Abs_time_token
  attr_accessor :time
  def initialize(time)
    @time = time
  end
end

class TimeScript
  def initialize
    @scriptParser = Parser.new("TimeScript") do
      @indent_stack = [0]

      #Pia Lötvedt's indentation code
      token(/\s*\n[ \t]*/) do |m|
        indent_size = m[/\s*\n\K[ \t]*/].gsub("\t", " "*4).size
        if indent_size == @indent_stack.last
          # Only newline if same as previous indent
          next :newline
        elsif indent_size > @indent_stack.last
          @indent_stack << indent_size
          # Newline followed by indent if larger than previous indent
          next [:newline, :indent]
        else
          dedents = [:newline]
          loop do
            @indent_stack.pop
            dedents << :dedent
            break if indent_size == @indent_stack.last
            # Ensures dedents goes back to an existant indendation level
            # (Maybe better with a custom error here, as its not a Ruby-SyntaxError)
            raise SyntaxError, "Bad indentation" if indent_size > @indent_stack.last
```

```
        end
        # Newline followed by a number of dedents if lesser than previous indent
        next dedents
      end
    end

    #token(/(\".*\"|\'.*\')/) {|m| String_token.new(m)} Denna rad är endast
    #utkommenterad för färgkorrigering. Den är inte utkommenterad i verkliga filen.
    token(/\s/)
    token(/\d{2}:\d{2}:\d{2}/) {|m| Abs_time_token.new(m)}
    token(/\d+[hms]/) {|m| m}
    token(/\d+\.\d+/) {|m| m.to_f}
    token(/\d+/) {|m| m.to_i}
    token(/\w+/) {|m| m}
    token(/(==|!=|<=|>=|\*\*|\+=|-=|\*=|\/=|<<)/) {|m| m}
    token(/./) {|m| m}

    start :program do
      match(:statement_list) {|a| Program_node.new(a).eval}
    end

    rule :statement_list do
      match(:statement_list, :line) {|a, b| Statement_list_node.new(a, b)}
      match(:line) {|a| Statement_list_node.new(a)}
    end

    rule :line do
      match(:statement, :newline)
      match(:statement)
    end

    rule :statement do
      match(:return)
      match(:break)
      match(:function)
      match(:at_stmt)
      match(:while_loop)
      match(:from_loop)
      match(:for_loop)
      match(:if_else_stmt)
      match(:assignment)
      match(:expr)
    end

    rule :param_list do
      match(:param_list, ',', String) {
        |a, _, b| List_node.new(a, String_node.new(b))}
      match(String) {|a| List_node.new(String_node.new(a))}
    end

    rule :arg_list do
      match(:arg_list, ',', :expr) {|a, _, b| List_node.new(a, b)}
      match(:expr) {|a| List_node.new(a)}
    end

    rule :function do
      match('def', String, '(', :param_list, ')', :indented_block) {
        |_, name, _, param_list, _, block|
        Assign_func_node.new(name, param_list, block)}
      match('def', String, '(', ')', :indented_block) {
        |_, name, _, _, block| Assign_func_node.new(name, param_list = nil, block)}
    end

    rule :function_call do
      match(/\w+/, '(', :arg_list, ')') {
```

```
      |name, _, arg_list| Run_func_node.new(name, arg_list)}
    match(/\w+/, '(', ')') {|name| Run_func_node.new(name)}
  end

  rule :at_stmt do
    match('at', :abs_time, :indented_block) {
      |_, abs_time, block| At_node.new(abs_time, block)}
  end

  rule :from_loop do
    match('from', :abs_time, 'to', :abs_time, 'each', :rel_time, :indented_block) {
      |_, abs_time_start, _, abs_time_end, _, step, block|
      From_node.new(abs_time_start, abs_time_end, block, step)}
    match('from', :abs_time, 'to', :abs_time, :indented_block) {
      |_, abs_time_start, _, abs_time_end, block|
      From_node.new(abs_time_start, abs_time_end, block)}
  end

  rule :for_loop do
    match('for', :rel_time, 'each', :rel_time, :indented_block) {
      |_, rel_time, _, step, block| For_time_node.new(rel_time, block, step)}
    match('for', :rel_time, :indented_block) {
      |_, rel_time, block| For_time_node.new(rel_time, block)}
    match('for', :assignment, ',', :comparison, ',', :assignment, :indented_block) {
      |_, assign, _, comp, _, step, block| For_node.new(assign, comp, step, block)}
  end

  rule :rel_time do
    match(/\d+h/, /\d+m/, /\d+s/) {|h, m, s|
      Rel_time_node.new(hours:h, minutes:m, seconds:s)}
    match(/\d+h/, /\d+m/) {|h, m| Rel_time_node.new(hours:h, minutes:m)}
    match(/\d+h/, /\d+s/) {|h, s| Rel_time_node.new(hours:h, minutes:nil, seconds:s)}
    match(/\d+m/, /\d+s/) {|m, s| Rel_time_node.new(minutes:m, seconds:s)}
    match(/\d+h/) {|h| Rel_time_node.new(hours:h)}
    match(/\d+m/) {|m| Rel_time_node.new(minutes:m)}
    match(/\d+s/) {|s| Rel_time_node.new(seconds:s)}
  end

  rule :abs_time do
    match(Abs_time_token) {|a| Abs_time_node.new(a.time)}
  end

  rule :while_loop do
    match('while', :boolean_expr, 'each', :rel_time, :indented_block) {
      |_, comp, _, step, block| While_node.new(comp, block, step)}
    match('while', :boolean_expr, :indented_block) {|_, comp, block|
      While_node.new(comp, block)}
  end

  rule :if_else_stmt do
    match(:if_stmt, :elsif_stmt, :else_stmt) {|a, b, c| If_else_node.new(a, b, c)}
    match(:if_stmt, :elsif_stmt) {|a, b| If_else_node.new(a, b)}
    match(:if_stmt, :else_stmt) {|a, b| If_else_node.new(a, b)}
    match(:if_stmt)
  end

  rule :if_stmt do
    match('if', :boolean_expr, :indented_block) {
      |_, boolean_expr, block| If_node.new(boolean_expr, block)}
  end

  rule :elsif_stmt do
    match('elsif', :boolean_expr, :indented_block, :elsif_stmt) do
      |_, boolean_expr, block, if_list|
      if_list << If_node.new(boolean_expr, block)
```

```
    end
    match('elsif', :boolean_expr, :indented_block) {
      |_, boolean_expr, block| [If_node.new(boolean_expr, block)]}
  end

  rule :else_stmt do
    match('else', :indented_block) {
      |_, block| [If_node.new(Boolean_node.new(true), block)]}
  end

  rule :indented_block do
    match(:newline, :indent, :statement_list, :newline, :dedent) {
      |_, _, stmt_list, _, _| stmt_list}
    match(:newline, :indent, :statement_list, :dedent) {
      |_, _, stmt_list, _| stmt_list}
    match(:newline, :indent, :statement_list) {|_, _, stmt_list| stmt_list}
  end

  rule :assignment do
    match(:var, '[', :arithmetic_expr, ']', '=', :expr) {
      |var, _, index, _, _, b| Assignment_index_node.new(var, index, b)}
    match(:var, '=', :expr) {|a, _, b| Assignment_node.new(a, b)}
    match(:var, :shorthand_op, :expr) {|a, op, b|
      Shorthand_assignment_node.new(a, op, b)}
  end

  rule :shorthand_op do
    match('+=')
    match('-=')
    match('*=')
    match('/=')
  end

  rule :expr do
    match(:nil)
    match(:input)
    match(:function_call)
    match(:list)
    match(:list_shorthand_add)
    match(:boolean_expr)
    match(String_token) {|st| String_node.new(st.string)}
  end

  rule :break do
    match('break') {Break_node.new()}
  end

  rule :return do
    match('return', :expr) {|_, expr| Return_node.new(expr)}
  end

  rule :nil do
    match('nil') {Nil_node.new()}
  end

  rule :input do
    match('input') {Input_node.new()}
  end

  rule :list do
    match('[', :arg_list, ']') {|_, list, _| list}
    match('[', ']') {List_node.new()}
  end

  rule :list_shorthand_add do
```

```
    match(:var, '<<', :expr) {|var, _, expr| List_shorthand_assign_node.new(var, expr)}
  end

  rule :boolean_expr do
    match(:boolean_expr, 'or', :boolean_term) {|a, _, b| Or_node.new(a, b)}
    match(:boolean_term)
  end

  rule :boolean_term do
    match(:boolean_term, 'and', :boolean_factor) {|a, _, b| And_node.new(a, b)}
    match(:boolean_factor)
  end

  rule :boolean_factor do
    match('not', :boolean_factor) {|_, a| Not_node.new(a)}
    match('true') {Boolean_node.new(true)}
    match('false') {Boolean_node.new(false)}
    match(:comparison)
  end

  rule :comparison do
    match(:arithmetic_expr, :comp_op, :arithmetic_expr) {
      |a, op, b| Comparison_node.new(a, op, b)}
    match(:arithmetic_expr)
  end

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

  rule :arithmetic_expr do
    match(:arithmetic_expr, '+', :arithmetic_term) {
      |a, _, b| Addition_node.new(a, b)}
    match(:arithmetic_expr, '-', :arithmetic_term) {
      |a, _, b| Subtraction_node.new(a, b)}
    match(:arithmetic_term)
  end

  rule :arithmetic_term do
    match(:arithmetic_term, '*', :arithmetic_factor) {
      |a, _, b| Multiplication_node.new(a, b)}
    match(:arithmetic_term, '/', :arithmetic_factor) {
      |a, _, b| Division_node.new(a, b)}
    match(:arithmetic_factor)
  end

  rule :arithmetic_factor do
    match(:arithmetic_primary, '**', :arithmetic_factor) {
      |a, _, b| Power_node.new(a, b)}
    match(:arithmetic_primary)
  end

  rule :arithmetic_primary do
    match('(', :boolean_expr, ')') {|_, a, _| a}
    match(:integer)
    match(:float)
    match(:list_var)
    match(:var) {|a| Get_var_node.new(a)}
  end
```

```
      rule :list_var do
        match(:list_var, '[', :arithmetic_expr, ']') {
          |var, _, index, _| Get_list_element_node.new(var, index)}
        match(:var, '[', :arithmetic_expr, ']') {
          |var, _, index, _| Get_list_element_node.new(var, index)}
      end

      rule :var do
        match(String) {|var| Var_node.new(var)}
      end

      rule :integer do
        match('-', Integer) {|neg, a| Integer_node.new(a, neg)}
        match(Integer) {|a| Integer_node.new(a)}
      end

      rule :float do
        match('-', Float) {|neg, a| Float_node.new(a, neg)}
        match(Float) {|a| Float_node.new(a)}
      end
    end
  end

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

  def run
    str = gets
    if done(str) then
      puts "Bye."
    else
      puts "=> #{@scriptParser.parse str}"
      run
    end
  end

  def parse_line(str)
    @scriptParser.parse str
  end

  def log(state = true)
    @scriptParser.logger.level = state ? Logger::DEBUG : Logger::WARN
  end
end

t = TimeScript.new()
Scope_handler.initialize
t.log(false)

if ARGV.length == 0
  puts "[TimeScript]"
  t.run
elsif ARGV.length == 1
  file = File.read(ARGV[0])
  t.parse_line(file)
else
  warn("WARNING, Timescript only accepts a single command line argument,
        arguments after the first one are ignored!")
end
```

5.2 nodes.rb

```
require 'time'
require_relative 'scope_handler.rb'

class Program_node
  def initialize(a)
    @program = a
  end

  def eval
    @program.eval
  end
end

class Statement_list_node
  def initialize(a, b = nil)
    @stmt_list = []
    if b.nil?
      @stmt_list << a
    else
      a.get_statements.each do |elem|
        @stmt_list << elem
      end
      @stmt_list << b
    end
  end

  def get_statements
    @stmt_list
  end

  def eval
    last_stmt = nil
    @stmt_list.each do |stmt|
      break if Scope_handler.get_break_flag

      if stmt.class == Return_node
        if Scope_handler.get_ongoing_function_calls == 0
          raise StandardError.new('Cannot return outside of a function!')
        end
        Scope_handler.set_return_flag(true)
        last_stmt = stmt.eval
        Scope_handler.set_return_value(last_stmt)
        break

      elsif stmt.class == Break_node
        if Scope_handler.get_ongoing_iterations == 0
          raise StandardError.new('Cannot break outside of a loop!')
        end
        Scope_handler.set_break_flag(true)
        break
      else
        last_stmt = stmt.eval
      end
    end
    last_stmt
  end
end

class List_node
  def initialize(a = nil, b = nil)
    @list = []
    if a
      if b.nil?
```

```
        @list << a
      else
        a.get_list.each do |elem|
          @list << elem
        end
        @list << b
      end
    end
  end

  def get_list
    @list
  end

  def unpack_list
    @list.each_with_index do |elem, index|
      if elem.class == List_node
        @list[index] = elem.unpack_list
      end
    end
  end

  def copy(list)
    tmp_list = Array.new
    list.each do |elem|
      tmp_list << elem
    end
    tmp_list
  end

  def evaluate_list(tmp_list)
    tmp_list.each_with_index do |elem, index|
      if elem.class == Array
        tmp_list[index] = evaluate_list(elem)
      else
        tmp_list[index] = elem.eval
      end
    end
    tmp_list
  end

  def eval
    #Unpacks list_nodes
    unpack_list()

    if @list.all? {|elem| elem.class == Integer}
      @list
    else
      tmp_list = copy(@list)
      evaluate_list(tmp_list)
    end
  end
end

class Run_func_node
  def initialize(name, arg_list = nil)
    @name = name
    @arg_list = arg_list
  end

  def eval
    Scope_handler.run_func(@name, @arg_list)
  end
end
```

```
class Assign_func_node
  def initialize(name, param_list, block)
    @name = name
    @param_list = param_list
    @block = block
  end

  def eval
    if @param_list.class == List_node
      @param_list = @param_list.eval
    end
    Scope_handler.assign_func(@name, @param_list, @block)
  end
end

class For_node
  def initialize(assign, comp, step, block)
    @assign = assign
    @comp = comp
    @step = step
    @block = block
  end

  def eval
    Scope_handler.add_variable_scope('iteration')
    @assign.eval
    while @comp.eval and (not Scope_handler.get_break_flag and not Scope_handler.
    get_return_flag)
      @block.eval
      @step.eval unless (Scope_handler.get_break_flag or Scope_handler.get_return_flag)
    end
    Scope_handler.pop_till_scope('iteration')
    Scope_handler.set_break_flag(false)
    #Scope_handler.pop_variable_scope
  end
end

class For_time_node
  def initialize(rel_time, block, step = nil)
    @rel_time = rel_time
    @block = block
    @step = step
  end

  def eval
    if @step and @step.eval > @rel_time.eval
      raise StandardError.new("Step can not be greater than total loop time!,
        Loop time: '#{@rel_time.eval}', Step: '#{@step.eval}'")
    end

    loop_for = Time.now.to_i + @rel_time.eval
    Scope_handler.add_variable_scope('iteration')
    while Time.now.to_i < loop_for
      @block.eval
      sleep(@step.eval) if @step
    end

    Scope_handler.pop_variable_scope
    Scope_handler.set_break_flag(false)
  end
end

class From_node
  def initialize(s, e, block, step = nil)
    @start = s
```

```
    @end = e
    @block = block
    @step = step
  end

  def eval
    @start = @start.eval
    @end = @end.eval

    if @end < @start
      @end += (60 * 60 * 24)
    end

    if @step and @step.eval > (@end.to_i - @start.to_i)
      raise StandardError.new("Step can not be greater than total loop time!,
        Loop time: '#{@rel_time}', Step: '#{@step}'")
    end

    while Time.now < @start
      sleep 1
    end

    Scope_handler.add_variable_scope('iteration')
    while Time.now < @end
      @block.eval
      sleep(@step.eval) if @step
    end
    Scope_handler.pop_variable_scope
    Scope_handler.set_break_flag(false)
  end
end

class At_node
  def initialize(abs_time, block)
    @abs_time = abs_time
    @block = block
  end

  def eval
    @abs_time = @abs_time.eval
    while Time.now < @abs_time
      sleep 1
    end
    Scope_handler.add_variable_scope('iteration')
    @block.eval
    @abs_time += (60 * 60 * 24)
    Scope_handler.pop_variable_scope
  end
end

class While_node
  def initialize(comp, block, step=nil)
    @comp = comp
    @block = block
    @step = step
  end

  def eval
    Scope_handler.add_variable_scope('iteration')
    while @comp.eval and (!Scope_handler.get_break_flag and !Scope_handler.get_return_flag)
      @block.eval
      sleep(@step.eval) if @step
    end
    Scope_handler.pop_variable_scope
    Scope_handler.set_break_flag(false)
```

```
  end
end

class If_node
  def initialize(boolean_expr, block)
    @boolean_expr = boolean_expr
    @block = block
  end

  def eval
    if @boolean_expr.eval
      Scope_handler.add_variable_scope('if')
      tmp = @block.eval
      Scope_handler.pop_variable_scope
      tmp
    end
  end
end

class If_else_node
  def initialize(*stmts)
    @stmts = stmts.flatten!
  end

  def eval
    @stmts.each do |stmt|
      @tmp = stmt.eval
      break if not @tmp.nil?
    end
    @tmp
  end
end

class Get_list_element_node
  def initialize(var, index)
    @var = var
    @index = index
  end

  def eval
    if @var.class == Get_list_element_node
      @var = @var.eval
    end

    if @var.class == Var_node
      list = Scope_handler.get_variable(@var.eval)
    else
      list = @var
    end

    if @index.eval > list.length - 1
      raise IndexError.new("Index out of bounds for #{@var.eval}. '\
                           'Was #{@index}, max #{list.length-1}")
    elsif @index.eval < 0
      raise IndexError.new("Negative index for #{@var.eval}. '\
                            Was #{@index}, must be zero or above")
    end
    list[@index.eval]
  end
end

class Get_var_node
  def initialize(var)
    @var = var
  end
```

```
  def get_name
    @var.eval
  end

  def eval
    Scope_handler.get_variable(@var.eval)
  end
end

class Assignment_node
  def initialize(var_name, expr)
    @var_name = var_name
    @expr = expr
  end

  def eval
    Scope_handler.assign_variable(@var_name.eval, @expr.eval)
  end
end

class Shorthand_assignment_node
  def initialize(var, op, expr)
    @var = var
    @op = op
    @expr = expr
  end

  def eval
    tmp = Scope_handler.get_variable(@var.eval)
    case @op
      when "+=" then Scope_handler.assign_variable(@var.eval, (tmp + @expr.eval))
      when "-=" then Scope_handler.assign_variable(@var.eval, (tmp - @expr.eval))
      when "*=" then Scope_handler.assign_variable(@var.eval, (tmp * @expr.eval))
      when "/=" then Scope_handler.assign_variable(@var.eval, (tmp / @expr.eval))
    end
  end
end

class Assignment_index_node
  def initialize(list, index, expr)
    @list = list
    @index = index
    @expr = expr
  end

  def eval
    tmp_list = Scope_handler.get_variable(@list.eval)
    tmp_list[@index.eval] = @expr.eval
    Scope_handler.assign_variable(@list.eval, tmp_list)
  end
end

class List_shorthand_assign_node
  def initialize(list, expr)
    @list = list
    @expr = expr
  end

  def eval
    tmp_list = Scope_handler.get_variable(@list.eval)
    if tmp_list.class == Array
      tmp_list << @expr.eval
      Scope_handler.assign_variable(@list.eval, tmp_list)
    else
```

```
      raise SyntaxError.new("Variable '#{@list.eval}' is not a list!")
    end
  end
end

class Break_node
  def eval
    #Scope_handler.pop_till_scope('iteration')
  end
end

class Return_node
  def initialize(return_value)
    @return_value = return_value
  end

  def eval
    @return_value.eval
  end
end

class Input_node
  def eval
    gets.chomp
  end
end

class Nil_node
  def eval
    nil
  end
end

class Or_node
  def initialize(a, b)
    @a = a
    @b = b
  end

  def eval
    @a.eval or @b.eval
  end
end

class And_node
  def initialize(a, b)
    @a = a
    @b = b
  end

  def eval
    @a.eval and @b.eval
  end
end

class Not_node
    def initialize(a)
    @a = a
    end

    def eval
    not @a.eval
    end
end
```

```
class Comparison_node
  def initialize(a, op, b)
    @a = a
    @op = op
    @b = b
  end

  def eval
    case @op
      when "<" then @a.eval < @b.eval
      when "<=" then @a.eval <= @b.eval
      when ">" then @a.eval > @b.eval
      when ">=" then @a.eval >= @b.eval
      when "==" then @a.eval == @b.eval
      when "!=" then @a.eval != @b.eval
    end
  end
end

class Boolean_node
  def initialize(a)
    @a = a
  end

  def eval
    @a
  end
end

class Addition_node
    def initialize(a, b)
    @a = a
        @b = b
    end

  def eval
    @a.eval + @b.eval
    end
end

class Subtraction_node
    def initialize(a, b)
    @a = a
    @b = b
    end

    def eval
    @a.eval - @b.eval
    end
end

class Multiplication_node
    def initialize(a, b)
    @a = a
        @b = b
    end

    def eval
    @a.eval * @b.eval
    end
end

class Division_node
    def initialize(a, b)
    @a = a
```

```
    @b = b
    end

    def eval
    @a.eval / @b.eval
    end
end

class Power_node
    def initialize(a, b)
    @a = a
    @b = b
    end

    def eval
    @a.eval ** @b.eval
    end
end

class Integer_node
  def initialize(a, neg = nil)
    if neg
      @int = -a
    else
      @int = a
    end
    end

  def eval
    @int
    end
end

class Float_node
  def initialize(a, neg = nil)
    if neg
      @float = -a
    else
      @float = a
    end
    end

    def eval
    @float
    end
end

class String_node
  def initialize(a)
    @string = a
    end

  def eval
    @string
    end
end

class Rel_time_node
  def initialize(hours:nil, minutes:nil, seconds:nil)
    @hours = hours ? hours.split('h')[0].to_i : 0
    @minutes = minutes ? minutes.split('m')[0].to_i : 0
    @seconds = seconds ? seconds.split('s')[0].to_i : 0
  end

  def eval
```

```
    (@hours * 3600 + @minutes * 60 + @seconds)
  end
end

class Abs_time_node
  def initialize(time)
    @time = time
  end

  def eval
    tmp = Time.now
    hours, minutes, seconds = @time.split(':')
    hours, minutes, seconds = hours.to_i, minutes.to_i, seconds.to_i
    skip_day = false

    #Checks if specified "at time" has already occured
    #and if so add an extra 24h to "at time"
    if hours < tmp.hour
      skip_day = true
    elsif hours == tmp.hour
      if minutes < tmp.min
        skip_day = true
      elsif minutes == tmp.min
        if seconds < tmp.sec
          skip_day = true
        end
      end
    end

    @time = Time.new(Time.now.year, Time.now.month, Time.now.day, hours, minutes, seconds)

    if skip_day == true
      @time += (60 * 60 * 24)
    end

    @time
  end
end

class Var_node
  def initialize(name)
    @name = name
    end

    def eval
        @name
    end
end
```

5.3 scope_handler.rb

```
require_relative 'functions.rb'

class Scope_handler
  @@variables = Array.new
  @@variables << Hash.new
  @@reserved_symbols = ['true', 'false', 'if', 'elsif', 'else', 'def', 'break',
                        'return', '==', '!=', '<=', '>=', '**', '+=', '-=',
                        '*=', '/=', '0DQdhzc2x8']
  @@functions = Hash.new
  @@reserved_functions = Array.new
  @@ongoing_function_calls = {ongoing: 0, return_value: nil, return_flag: false}
  @@ongoing_iterations = {ongoing: 0, break_flag: false}

  #Adds all predefined functions to @@reserved_functions
  def Scope_handler.initialize
    IO.foreach($install_directory + 'functions.rb') do |line|
      if line =~ /def\s(\w+)/
        @@reserved_functions << $1
      end
    end
  end

  def Scope_handler.get_ongoing_function_calls
    @@ongoing_function_calls[:ongoing]
  end

  def Scope_handler.get_return_flag
    @@ongoing_function_calls[:return_flag]
  end

  def Scope_handler.get_return_value
    @@ongoing_function_calls[:return_value]
  end

  def Scope_handler.set_return_flag(boolean)
    @@ongoing_function_calls[:return_flag] = boolean
  end

  def Scope_handler.get_ongoing_iterations
    @@ongoing_iterations[:ongoing]
  end

  def Scope_handler.get_break_flag
    @@ongoing_iterations[:break_flag]
  end

  def Scope_handler.set_break_flag(boolean)
    @@ongoing_iterations[:break_flag] = boolean
  end

  def Scope_handler.set_return_value(value)
    @@ongoing_function_calls[:return_value] = value
  end

  def Scope_handler.add_variable_scope(type, scope = nil)
    @@ongoing_iterations[:ongoing] += 1
    new_scope = Hash.new

    #Copies over each variable from the last scope to the new one
    @@variables.last().each do |key, value|
      new_scope[key] = value
    end
    new_scope['0DQdhzc2x8'] = type
```

```
    @@variables << new_scope

    #Adds incoming variables to new scope
    if scope
      scope.each do |key, value|
        @@variables.last()[key] = value
      end
    end
  end

  def Scope_handler.add_function_scope(scope = nil)
    @@ongoing_function_calls[:ongoing] += 1
    new_scope = Hash.new
    new_scope['0DQdhzc2x8'] = "func"

    #Adds incoming variables to the new scope
    if scope
      scope.each do |key, value|
        new_scope[key] = value
      end
    end
    @@variables << new_scope
  end

  def Scope_handler.pop_variable_scope
    #Pop the last scope and transfer the value of all variables whose name
    #also exists in the new last scope
    old_scope = @@variables.pop()
    @@variables.last().each do |key, value|
      if @@variables.last().key?(key) and old_scope[key] != value and key != '0DQdhzc2x8'
        @@variables.last()[key] = old_scope[key]
      end
    end
    old_scope
  end

  def Scope_handler.pop_function_scope
    @@variables.pop()
  end

  def Scope_handler.pop_till_scope(type)
    #Pops scopes until the new last scope equals 'type' or until there is
    #only one scope left.
    loop do
      if @@variables.length == 1
        break
      end
      if @@variables.last()['0DQdhzc2x8'] == 'func'
        old_scope = Scope_handler.pop_function_scope
        @@ongoing_function_calls[:ongoing] -= 1
      else
        old_scope = Scope_handler.pop_variable_scope
        @@ongoing_iterations[:ongoing] -= 1
      end

      if old_scope['0DQdhzc2x8'] == type
        break
      end
    end
  end

  def Scope_handler.assign_variable(name, data)
    if not @@reserved_symbols.include?(name)
      @@variables.last()[name] = data
    else
```

```
      raise SyntaxError.new("Cannot assign to reserved symbols!")
    end
  end

  def Scope_handler.get_variable(name)
    if @@variables.last().include?(name)
      @@variables.last()[name]
    else
      raise SyntaxError.new("Variable '#{name}' does not exist!")
    end
  end

  def Scope_handler.assign_func(name, param_list, block)
    @@functions[name] = {parameters: param_list, block: block}
  end

  def Scope_handler.run_func(name, arg_list=nil)
    #If the called function is a predefined/built-in function of the language
    if @@reserved_functions.include?(name)
      if arg_list.class == List_node
        if arg_list.get_list[0].class == Get_var_node and arg_list.get_list.length != 1
          var_name = arg_list.get_list[0].get_name
          arg_list = arg_list.eval
          tmp = eval "#{name}(#{arg_list})"
          Scope_handler.assign_variable(var_name, tmp)
        else
          arg_list = arg_list.eval
          eval "#{name}(#{arg_list})"
        end
      end

    #If the called function is included in @@functions
    elsif @@functions.include?(name)
      func = @@functions[name]

      #If the function has parameters combine the function parameters
      #with the given arguments and add a function scope with the
      #param_arg_list, else create an empty function scope.
      if func[:parameters]
        param_list = func[:parameters]

        if arg_list.class == List_node
          arg_list = arg_list.eval
        end

        if param_list.length != arg_list.length
          raise SyntaxError.new("Wrong amount of arguments. '\
          'Got #{arg_list.length}, should be #{param_list.length}")
        end

        param_arg_list = Hash.new
        param_list.each_with_index do |v, i|
          param_arg_list[v] = arg_list[i]
        end

        Scope_handler.add_function_scope(param_arg_list)
      else
        Scope_handler.add_function_scope()
      end

      #Run the function and then pop all scopes to the closest function
      #scope including the function scope itself
      func[:block].eval
      Scope_handler.pop_till_scope('func')
```

```
      #Gets and returns the function's return value
      @@ongoing_function_calls[:return_flag] = nil
      tmp = @@ongoing_function_calls[:return_value]
      @@ongoing_function_calls[:return_value] = nil
      tmp
    else
      raise SyntaxError.new("Function '#{name}' does not exist!")
    end
  end
end
```

5.4 functions.rb

```
def print(item = "\n")
  puts "#{item[0]}"
end

def remove_at(arg_list)
  if arg_list[1] > arg_list[0].length - 1
    raise IndexError.new("Index out of bounds")
  elsif arg_list[1] < 0
    raise IndexError.new("Negative index. Was #{arg_list[1]}, '\
                         'must be zero or above")
  end
  arg_list[0].delete_at(arg_list[1])
  arg_list[0]
end

def len(list)
  list[0].length
end

def int(value)
  raise ArgumentError.new("Wrong amount of arguments. Got #{value.length}, '\
                          'should be 1") if value.length != 1
  begin
    Integer(value[0])
  rescue ArgumentError
    raise ArgumentError.new("Cannot convert value '#{value[0]}' to an integer!")
  end
end

def string(value)
  raise ArgumentError.new("Wrong amount of arguments.'\
  ' Got #{value.length}, should be 1") if value.length != 1
  String(value[0])
end
```

5.5 rdparse.rb

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

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

require 'logger'

class Rule

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

  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    @logger = parser.logger
    # The name of the expressions this rule matches
    @name = name
    # We need the parser to recursively parse sub-expressions occurring
    # within the pattern of the match objects associated with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #   match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.
  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
    if pattern[0] == @name
      pattern.shift
      @lrmatches << match
    else
      @matches << match
    end
  end

  def parse
    # Try non-left-recursive matches first, to avoid infinite recursion
    match_result = try_matches(@matches)
    return nil if match_result.nil?
    loop do
      result = try_matches(@lrmatches, match_result)
      return match_result if result.nil?
      match_result = result
    end
  end

  private

  # Try out all matching patterns of this rule
  def try_matches(matches, pre_result = nil)
    match_result = nil
    # Begin at the current position in the input string of the parser
    start = @parser.pos
```

```
    matches.each do |match|
      # pre_result is a previously available result from evaluating expressions
      result = pre_result.nil? ? [] : [pre_result]

      # We iterate through the parts of the pattern, which may be e.g.
      #   [:expr,'*',:term]
      match.pattern.each_with_index do |token,index|

        # If this "token" is a compound term, add the result of
        # parsing it to the "result" array
        if @parser.rules[token]
          result << @parser.rules[token].parse
          if result.last.nil?
            result = nil
            break
          end
          @logger.debug("Matched '#{@name} = #{match.pattern[index..-1].inspect}'")
        else
          # Otherwise, we consume the token as part of applying this rule
          nt = @parser.expect(token)
          if nt
            result << nt
            if @lrmatches.include?(match.pattern) then
              pattern = [@name]+match.pattern
            else
              pattern = match.pattern
            end
            @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #{pattern.
    inspect}'")
          else
            result = nil
            break
          end
        end
      end
      if result
        if match.block
          match_result = match.block.call(*result)
        else
          match_result = result[0]
        end
        @logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#{@name}' and
    generated '#{match_result.inspect}'") unless match_result.nil?
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
      end
    end

    return match_result
  end
end

class Parser

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

  class ParseError < RuntimeError
  end

  def initialize(language_name, &block)
    @logger = Logger.new(STDOUT)
```

```
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name = language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @tokens = []
    @string = string.clone
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
      raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |tok|
        match = tok.pattern.match(string)
        # The regular expression of a token has matched the beginning of 'string'
        if match
          @logger.debug("Token #{match[0]} consumed")
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
          true
        else
          # this token pattern did not match, try the next
          false
        end # if
      end # raise
    end # until
    @tokens.flatten!
  end

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

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

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

  # Return the next token in the queue
  def expect(tok)
    return tok if tok == :empty
    t = next_token
    if @pos - 1 > @max_pos
      @max_pos = @pos - 1
```

```
      @expected = []
    end
    return t if tok === t
    @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
    return nil
  end

  def to_s
    "Parser for #{@language_name}"
  end

  private

  LexToken = Struct.new(:pattern, :block)

  def token(pattern, &block)
    @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
  end

  def start(name, &block)
    rule(name, &block)
    @start = @rules[name]
  end

  def rule(name,&block)
    @current_rule = Rule.new(name, self)
    @rules[name] = @current_rule
    instance_eval &block
    @current_rule = nil
  end

  def match(*pattern, &block)
    @current_rule.send(:match, *pattern, &block)
  end
end
```