LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Termer

Författare

Alexander Jonsson, alejo720@student.liu.se
Gustav Persson, gussv375@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2017
Version 1.2

2017-05-17

Innehåll

1 Revisionshistorik **2**

2 Inledning **2**

3 Användarhandledning **3**
3.1 Viktigt att veta . . . 4
3.2 Datatyper . . . 4
3.2.1 Heltal . . . 4
3.2.2 Flyttal . . . 4
3.2.3 Strängar . . . 4
3.2.4 Boolsk . . . 4
3.2.5 Samling . . . 5
3.2.6 null-värde . . . 5
3.3 Variabler . . . 6
3.4 Aritmetiska uttryck . . . 6
3.5 Tilldelning till variabel . . . 7
3.6 Utmatning/Inmatning . . . 7
3.6.1 Utmatning . . . 7
3.6.2 Inmatning . . . 8
3.7 Villkor . . . 9
3.7.1 Villkorsuttryck . . . 9
3.7.2 Villkorssatser . . . 10
3.8 start/stop-block . . . 11
3.9 Loopar . . . 12
3.10 Oändliga loopar . . . 12
3.10.1 Medans-loopar . . . 12
3.10.2 Räkneloopar . . . 12
3.10.3 För-loopar . . . 13
3.11 Funktioner . . . 13
3.11.1 Funktioner utan parametrar . . . 14
3.11.2 Funktioner med parametrar . . . 14
3.11.3 Funktioner med returvärde . . . 14
3.11.4 Funktioner med förvalt värde som parameter . . . 15
3.12 Klasser . . . 15

4 Systemdokumentation **15**
4.1 Grammatik . . . 15
4.2 Systemets olika delar . . . 18
4.2.1 Lexikalisk analys . . . 18
4.2.2 Parsning . . . 18
4.2.3 Evaluering . . . 18
4.3 Klasser och dess relationer . . . 18
4.3.1 Program . . . 18
4.3.2 Scope . . . 18
4.3.3 Variable . . . 18
4.3.4 Expression . . . 18
4.3.5 Condition . . . 19
4.3.6 Assignment . . . 19
4.3.7 AssignmentIncrement . . . 19
4.3.8 StringConcatenation . . . 19

4.3.9 Output . . . 19
4.3.10 Input . . . 20
4.3.11 Block . . . 20
4.3.12 If . . . 20
4.3.13 ConditionStatement . . . 20
4.3.14 Restart . . . 20
4.3.15 Start . . . 20
4.3.16 Return . . . 20
4.3.17 Function . . . 21
4.3.18 FunctionCall . . . 21
4.3.19 Loop . . . 21
4.3.20 LoopWhile . . . 21
4.3.21 LoopFor . . . 21
4.3.22 LoopArray . . . 22
4.3.23 ArrayCall . . . 22
4.3.24 Null . . . 22
4.4 Val av representation av syntaxträd . . . 22
4.5 Algoritmer . . . 22
4.6 Kodstandard . . . 22
4.7 Användning och installation . . . 22

5 Erfarenheter och reflektion 22
5.1 Vad gick lättare och vad var svårare än vi trodde? . . . 22
5.2 Hur mycket kunde vi följa språkspecifikationen? . . . 22

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.2	Användarmanual, delar av systemdokumentation samt delar av reflektionsdokumentet	2017-05-16
1.1	Jobbat på användarmanual	2017-05-15
1.0	Första version, inledning och struktur på dokument	2017-05-12

2 Inledning

Detta är ett projekt på programmet Innovativ Programmering i kursen TDP019 Projekt: datorspråk. I detta projekt har vi fått uppdraget att skapa ett eget programmeringsspråk. Vi har skapat ett imperativt programmeringsspråket vi kallar för 'Termer'.

I vardagligt språk används det många programmeringstermer som inte används som nyckelord inuti programmeringsspråken, så som 'getter', 'setters', 'constructor'. Så är det dock inte i detta programmeringsspråk. Dessa termer används som nyckelord i detta programmeringsspråk. Därför har vi valt att kalla programmeringsspråket för just 'Termer' (uttalas på Engelska).

3 Användarhandledning

Termer är ett språk relativt lätt att lära sig men svårare att bemästra. Det innehåller en stor del av funktionalitet från andra språk men har även nya funktioner. Detta leder till att språket kommer locka både nybörjare och erfarna programmerare. Nybörjarna kommer snabbt slängas in i programmeringsvärlden och dess termer medan de mer erfarna kommer uppskatta programmeringsspråkets flexibilitet och nya funktioner.

3.1 Viktigt att veta

- I Termer omringas block med måsvingar - Uttryck som inte innehåller block måste alltid slutas med ett semikolon. - Kommentarer: I termer går det skapa en kommentarer genom två stycken snedstreck efter varandra. Detta kommer göra så att allt på samma rad efter snedstrecken inte kommer att tolkas som kod utan istället ignoreras.

```
// Detta är en kommentar
a = "detta är inte en kommentar och tollkas som kod!";
// Kommentarer behöver inte semikolon.
// a = "detta är en kommentar och tolkas som kod!";
```

3.2 Datatyper

I Termer, som i princip alla språk, finns det olika typer av data, kallat datatyper. Dessa datatyper finns inbyggt i Termer:

3.2.1 Heltal

Heltal skrivs som en siffra utan decimal, exempelvis:

```
0;
100;
```

3.2.2 Flyttal

Flyttal är samma sak som ett decimaltal

```
5.5;
3.1415926535;
```

3.2.3 Strängar

En sträng är en samling karaktärer. För att skapa en sträng används dubbla citationstecken. Mellan dessa citationstecken går det skriva nästan vilka tecken som helst (förutom ett till citationstecken). Det går att sätta ihop strängar genom att ha punkt mellan strängarna (punkten får dock inte vara precis första strängen), exempelvis:

```
"Hello";
"Termer is awesome";
"38 is awesome number!";
//Ihopsättning av en sträng
"Hello" . "World" . !;
//Nedan fungerar inte eftersom strängen slutar innan "Name"
"my name is "Name"";
```

3.2.4 Boolsk

Ett boolskt värde (Engelska: Boolean) är ett värde som antingen är sant eller falskt, exempelvis:

```
true;
false;
```

3.2.5 Samling

En samling (engelska: Array) innehåller en mängd data. Datan skrivs mellan hakparanteser och datan separeras med kommatecken, exempelvis:

```
[1,2,3,4,5,6,7,8,9,10];
["A", "E", "I", "O", "U", "Y"];
//Notera att det går att blanda datatyper
[1, "two", "three", 4, 5, 6, "seven"];
```

3.2.6 null-värde

Ett null-värde är egentligen inget värde utan betyder att det inte finns ett värde. För att ange nullskrivs det precis som det heter, null".

```
nothing = null;
```

3.3 Variabler

Variabler är enkelt beskrivet ett namn tilldelat data. Namnet på variabeln kan dock inte vara vad som helst utan första tecknet måste vara en (engelsk) bokstav eller ett understreck och följande tecken detsamma fast siffror tillåts, exempelvis:

```
//Giltiga variabelnamn
x
name

_
__number
ice_cream
HELLO
five5

//Ogiltiga variabelnamn
4
1one
Hello!
?doesthiswork?
```

För att skapa en variabel måste den tilldelas (se nedan)

3.4 Aritmetiska uttryck

Aritmetiska uttryck är i princip matematiska uttryck och kan innehålla operatorerna: + - * / %.
Ett aritmetiskt uttryck kan exempelvis se ut på följande sätt:

```
>>1+1;
==2
>>5-1;
==4
>>10*5;
==50
>>20/4;
==5
>>2^3;
==8
>>10%2;
==0
>>10%3;
==1
>>5+5*5;
==30
>>5^(1-1);
==1
>>5*-1;
==-5
>>5^2/2;
==12.5
>>100^0.5;
==10.0
```

3.5 Tilldelning till variabel

För att tilldela en variabel används följande syntax:
<variable name> <assignment_operator> <data/expression>;

Det finns följande tilldelningsoperatorer: '=' '+=' '-=' '*=' '/=' '%=' '^='. För att först skapa en variabel måste den tilldelas med operatorn '='. Resterande operatorer gör i princip så att variabeln slängs in i uttrycket.

Alla variabler måste ha en datatyp men lägg märke till att I Termer behövs det inte ange vilken datatyp variabeln ska ha utan datatypen bestäms beroende på vilken typ av data som tilldelas.

Det är vanligt att vilja öka eller minska en variabel med 1. Därför finns det ett smidigt sätt att göra detta, se syntax:
<variable name> <increment_operator>;
En tillväxtsoperator är antingen '++' eller '–' beroende på om det är ökning respektive minskning som vill åstadkommas.

Exempel för tilldelning av variabel:

```
>>x = 0;
0
>>x++;
==1
>>x+=9;
==10
>>x/=2;
==5
>>x%=2;
==1
>>x*=2;
==2
>>x^=3;
==8
>>x-=7;
==1
>>x--;
==0
```

3.6 Utmatning/Inmatning

Utmatning och inmatning är till för att skriva ut i terminalen och hämta användarinmatning från terminalen.

3.6.1 Utmatning

Att skriva ut i terminalen kan vara bra för användarvänlighetens skull och för testning. Syntaxen för utmatning ser ut så här:
output (<expression>);
eller...
output (<condition>);
Inuti paranteserna måste det vara antingen ett uttryck, data eller en villkorssats. Det går att skriva ut vilken datatyp som helst.

3.6.2 Inmatning

Vid inmatning väntar programmet på att användaren skriver något i terminalen för att sedan lagra resultatet i en variabel. Syntaxen för inmatning ser ut så här:
<variable name> <assignment_operator> input ( )
eller...
<variable name> <assignment_operator> input (<expression>)
Inuti paranteserna går det skriva ett uttryck. Detta uttryck kommer skrivas ut innan användaren inmatar något. Det kan t.ex. vara smidigt att skriva ut vad användaren ska mata in.
Nedan är exempel på inmatning:

```
username = input("Please input username: ");
password = input("Please input password: ");
output("Press any key twice to continue);
any_key = input();
any_key = input();
```

3.7 Villkor

Ett villkor är ett uttryck eller data som antingen evalueras till sant eller falskt. Ett villkor ger alltså alltid tillbaks ett boolsk värde.

3.7.1 Villkorsuttryck

Ett villkorsuttryck innehåller logiska operatorer och/eller jämförelseoperatorer. Uttrycken i ett villkorsuttryck kan även negeras med negationsoperatorer.
Logiska operatorer: '&/and' '|/or'
Jämförelseoperatorer: '!=/not equals' '==/equals' '<=' '>=' '<' '>'
Negationoperatorer: '!/not'

Lägg märkte till att det i vissa fall finns två olika sätt att skriva samma operator, där ett av sätten använder sig av ord istället vilket gör det mer. Detta är en av anledningarna till att Termer är ett flexibelt språk som både lockar nybörjare och erfarna programmerare.
Nedan är exempel på villkorsuttryck:

```
>>true;
==true
>>false;
==false
>>true & true;
==true
>>true and false;
==false
>>true | false;
==true
>>false or false;
==false
>>10 < 20;
==true
>>20 > 30;
==false
>>5 >= 5
==true
>>5 <= 4
==false
>>5 equals 5 and 4 == 4
==true
>>5 != 4
==true
>>"hello" not equals "hello"
==false
>>not true
==false
>>!false
==true
Notera att 0 och null evalueras till falskt
>>false or 0
==false
>>false or null
==false
```

3.7.2 Villkorssatser

Tillsammans med villkorsuttryck går det skapa villkorssatser. En villkorssats är en block med kod som endast körs om ett specifikt villkorsuttryck evalueras till sant. Vi kallar den för ”if-sats och så här ser den syntaxen ut:
’if’ ’(’ <condition> ’)’ <block>

Om inte villkorsuttrycket evalueras till sant går det göra så ett annat block körs istället. Vi kallar den för ”else-sats och en ”else-sats måste skrivas efter en villkorssats. Så här den syntaxen ut:
’else’ <block>

Det går även kombinera dessa två till en lång sekvens av block. Så om villkorsuttrycket inte evalueras till sant går det göra så ett ett annat block körs om ett annat villkorsuttryck evalueras till sant. Vi kallar den för en ”elseif-sats och den måste skrivas efter en ”if-sats och innan en ”else-sats. Så här ser denna syntax ut: ’elseif’ (<condition>) <block>

Nedan är exempel på villkorssatser:

```
x = false;
if(not x)
{
    output("This will get printed");
}

x = null;
if(x and x)
{
    output("This will not be printed");
}

//Kolla om ett tal är jämnt eller udda
x = 5;
if (x % 2 == 0)
{
    output(x . " is not even");
}
else
{
    output(x . " is odd");
}

role = input("Select your role: ");
if(role == "admin")
{
    output("You are an admin");
}
elseif(role == "user")
{
    output("You are an user");
}
else
{
    output(role . " is not a valid role");
}

dir = input("Select direction");
```

```
if (dir == "up")
{
    output("You moved up");
}
elseif (dir == "down")
{
    output("You moved down");
}
elseif (dir == "left")
{
    output("You moved to the left");
}
elseif (dir == "right")
{
    output("You moved to the right");
}
```

3.8 start/stop-block

En av de bästa funktionerna med detta språk är start/stop-blocket. Mellan orden 'start' och 'stop' skrivs koden som vanligt men om ordet 'restart' hittas kommer variablerna inuti blocket sparas och koden börja ifrån 'start' igen. Denna funktion inspirerades av onödig upprepning av kod vi stött på i andra språk. Tänk dig ett scenario där du vill läsa in ett namn, kolla om namnet är giltigt, om inte skriva ut felmeddelande och fråga om ett nytt namn. I C++ skulle det kunna åstadkommas med denna kod:

```
cout >> "Please input age: ";
age << cin;
while ( is_valid(age) == false )
{
    cout >> "Wrong input";
    cout >> "Please input age: ";
    age << cin;
}
```

Eller denna...

```
do
{
    cout >> "Please input age: ";
    age << cin;
    if ( is_valid(age) == false)
    {
        cout >> "Wrong input";
    }

} while (is_valid(age) == false);
```

Som du ser krävs det återupprepning av kod i båda fallen. Med start/stop-block går det skriva så här:

```
start
    age = input("Please input age: ");
    if ( is_valid(age) == false)
    {
        output("Wrong input");
        restart
    }
```

```
stop
```

Som du ser krävs det ingen återupprepning av kod med start/stop-block.

3.9 Loopar

En loop är en block med kod som evalueras ett antal gånger beroende på olika faktorer. Termer är loopar både avancerade och flexibla. Detta beror på att det finns så många olika sätt att skriva loopar och alla sätt har liknande syntax.

3.10 Oändliga loopar

Oändliga loopar som hörs på namnet kör sitt block oändligt många gånger.
Syntaxen för oändliga loopar ser ut så här:
loop <block>
Nedan är ett kodexempel på oändliga loopar:

```
loop
{
    output("This message will never end and hunt you forever, muhaha");
}
```

3.10.1 Medans-loopar

Denna loop kommer att köra blocket så länge ett specifikt villkor evaluerar till sant. Syntaxen ser ut så här:
loop (<condition>) <block>

Nedan är ett exempel på en medans-loop:

```
//Loop until condition is false
name = "";
valid = false;
loop (valid == false)
{
    name = input("Input name please: ");
    if (name == "")
    {
        output(name . " is not a valid name");
    }
    else{valid = true;}
}
output("Your awesome name is " . name);
```

3.10.2 Räkneloopar

Räknligteloopar är de loopar som skiljer sig mest från andra språk. (Räkneloopar körs ett specifikt antal gånger precis som de flesta loopar men skillnaden är att det går att det själv går att ange direkt antal gånger istället för att antal gånger ska bero på något villkor (som i for-loopar i andra språk))? Det finns tre olika räkneloopar, se syntax:
1. loop (<expression>) <block>
2. loop (<expression> , <variable name>) <block>
3. loop (<expression> , <variable name> , <assignment>) <block>

1. Denna loop kommer att köras lika många gånger som anges i parametern, exempelvis:

```
//Räkna från 1 till 20
x = 0;
loop (20)
{
    x++;
    output(x);
}
```

2. Denna loop kommer att köras lika många gånger som anges i parametern, men variabeln som anges kommer auomatiskt tilldelas noll och öka med ett efter varje iteration, exempelvis:

```
//Räkna från 0 till 19
loop (20, i)
{
    output(i);
}
```

3. Denna loop är likadan som nummer 2 fast det går att ange en egen tilldelning till den angivna variabeln, exempelvis:

```
//Räkna ut summan av alla tal mellan 0 och 18
sum = 0;
loop (10, i, i+=2)
{
    sum+=i;
}
//0+2+4+6+8+10+12+14+16+18=90
output(sum);
```

3.10.3 För-loopar

Denna loop är exakt som en for-loop i vilket annat språk som helst. Det här är vad som händer i en for-loop:
1. En variabel tilldelas
2. Blocket körs
3. En variabeln ändrar sitt värde (föredraget samma variabel som tilldelades i steg 1).
4. Ett villkorsuttryck evalueras. Om villkorsuttrycket evalueras till sant gå till steg 2.

Nedanför är ett exempel på for-loopar:

```
//Count from 2 to 5
sum = 0;
loop (i=2, i<=5, i++)
{
    sum = sum + i;
}
//2 + 3 + 4 + 5 = 14
output(sum);
```

3.11 Funktioner

En funktion är ett namngett block med kod som helst ska ha l en specieluppgift. Detta block med kod går sedan att kalla på för att köra blocket med kod. En funktion kan även innehålla parametrar. En parameter

är en variabel som tilldelas i samband med funktionsanropet.
Så här ser syntaxen för en funktionsdeklaration ut: fun <fun_name> (<parameter_list>) <function_block>

För att sedan kalla på en funktion används följande syntax:
<function_name> (<function_call_parameters> ')'

3.11.1 Funktioner utan parametrar

Funktioner behöver inte alltid ha parametrar utan det går även deklarera en funktion enligt följande syntax:
fun <function_name> ( ) <function_block>

Nedan är ett exempel på en funktion utan några parametrar:

```
fun greet()
{
    output("Hello");
}
greet();
```

3.11.2 Funktioner med parametrar

Parametrarna i en funktionsdeklaration skrivs som variabelnamn separerade med kommatecken, exempelvis:

```
fun print_time(hour, min, sec)
{
    output(hour . ":" . min . ":" . sec . ":");
}
//Will output "23:14:38"
print_time(23, 14, 38);

fun greet(name)
{
    output("Hello " . name . "!");
}
greet("Alexander");
greet("Gustav");
```

3.11.3 Funktioner med returvärde

En ordet returnstöts på returnerar funktionen värdet som står efter returnöch slutar evaluera funktionen.
Syntaxen ser ut så här:
return <expression>;
Detta värde kan sedan användas t.ex. i uttryck, se exempel nedan:

```
fun sum(a, b)
{
    return a+b;
}

//Will output 100
output(sum(5,5) * 10);

x = sum(sum(8,2), sum(6,4));
//Will output 20
output(x);
```

```
fun valid_username(username)
{
    if (username not equals "This username is taken")
    {
        return true;
    }
    return false;
}

if (valid_username("Codelover1338"))
{
    output("Registered");
}

if (valid_username("This username is taken"))
{
    output("The username is already taken, try again");
}
```

3.11.4 Funktioner med förvalt värde som parameter

Det går att ha parametrar med förvalt värde vilket betyder att det inte är obligatoriskt att ange just den parametern vid funktionsanrop. Syntaxen ser ut föjande:
<variable_name> = <data>
Om inte variabeln anges i funktionsanropet kommer den att få värdet den tilldelades i funktionsdeklarationen. Se nedan för exempel på funktioner med förvalt värde som parameter:

```
fun eat(kilos = 1, food = "Pizza")
{
    output("You just ate " . kilos . " of " . food);
}
//You just ate 1 kilos of pizza
eat();
//You just ate 10 kilos of pizza
eat(10);
//You just ate 100 kilos of carrot (you must be a horse)
eat(100, "Carrot");
```

3.12 Klasser

Tyvärr finns inga klasser i vårt språk. De kommer i version 2.0!

4 Systemdokumentation

Översiktlig beskrivning:

4.1 Grammatik

```
Note: Regex expressions are surrounded with '/'-characters
--------------------START PROGRAM HERE--------------------
<program>::=<statements>
<program>::=EMPTY
```

```
<statements>::=<function_declaration> <statements>
            |<function_declaration>
            |<compound_statement> <statements>
            |<compound_statement>
            |<simple_statement> ';' <statements>
            |<simple_statement> ';'

<simple_statement>::=<return_statement>
                 |<IO>
                 |<assignment>
                 |<expression>

<compound_statement>::='restart'
                  |<start_statement>
                  |<loop_statement>
                  |<condition_statement>

<return_statement>::='return' <expression>

<IO>::=output '(' <expression> ')'
     |output '(' <condition> ')'
     |<var> <assignment_operator> 'input' '(' ')'
     |<var> <assignment_operator> 'input' '(' <expression> ')'

<assignment>::=<array_call> <assignment_operator> <expression>
            |<var> <assignment_operator> <expression>
            |<array_call> <increment_operator>
            |<var> <increment_operator>

<condition>::=<condition> <logical_operator> <comparison>
           |<comparison>
<comparison>::=<comparison> <comparison_operator> <condition_data>
            |<condition_data>
<condition_data>::=<negation_operator> <condition>
               |<expression>
               |'(' <condition> ')'

<expression>::=<arithmetic_expression>
            |<expression> /(\s\.\s|\s\.)/ <expression>
<arithmetic_expression>::=<arithmetic_expression> <add_operator> <multi_expression>
            |<multi_expression>
<multi_expression>::=<multi_expression> <multi_operator> <simple_expression>
                 |<simple_expression>
<simple_expression>::=<function_call>
                  |<array_call>
                  |<data>
                  |<var>
                  |'-' <simple_expression>
                  |'+' <simple_expression>
                  |'(' <arithmetic_expression> ')'

<function_call>::=<var> '(' <function_call_parameters> ')'
              |<var> '(' ')'
<function_call_parameters>::=<expression> ',' <function_call_parameters>
                         |<expression>
```

```
<array_call>::=<var> '[' <expression> ']'

<data>::=/"[^\"]*"/
       |FLOAT
       |INTEGER
       |ARRAY
       |'null'
       |'true'
       |'false'
       |<array>

<array>::='[' <array_data> ']'
<array_data>::=<expression> ',' <array_data>
            |<expression>

<var>::=/^[A-Za-z_][A-Za-z\d_]*/

<start_statement>::='start' <start_block> 'stop'
<start_block>::=<statements>

<block>::='{' <statements> '}'

<loop_statement>::='loop' <block>
                |'loop' '(' <expression> ')' <block>
                |'loop' '(' <expression> ',' <var> ')' <block>
                |'loop' '(' <expression> ',' <var> ',' <assignment> ')' <block>
                |'loop' '(' <condition> ')' <block>
                |'loop' '(' <assignment> ',' <condition> ',' <assignment> ')' <block>

<condition_statement>::=<if_statement> <elseif_statement> <else_statement>
                    |<if_statement> <elseif_statement>
                    |<if_statement> <else_statement>
                    |<if_statement>
<if_statement>::='if' '(' <condition> ')' <block>
<elseif_statement>::='elseif' '(' <condition> ')' <block> <elseif_statement>
                  |'elseif' '(' <condition> ')' <block>
<else_statement>::='else' <block>

<function_declaration>::='fun' <var> '(' ')' <block>
                 |'fun' <var> '(' <parameter_list> ')' <block>
<parameter_list>::=<var> ',' <parameter_list>
                |<default_parameter> ',' <parameter_list>
                |<default_parameter>
                |<var>
<default_parameter>::=<var> '=' <data>

<fun_block>::='{' <statements> <return_statement> <statements> '}'
           |'{' <return_statement> '}'
           |<block>

<add_operator>::='+'|'-'
<multi_operator>::='^'|'*'|'/'|'%'

<assignment_operator>::='='|'+='|'-='|'*='|'/='|'%='|'^='
<increment_operator>::='++'|'--'
```

```
<logical_operator>::='&'|'and'| '|' |'or
<comparison_operator>::='!='|'not' 'equals'|'=='|'equals'|'<='|'>='|'<'|'>'
<negation_operator>::='not'|'!'
```

4.2 Systemets olika delar

4.2.1 Lexikalisk analys

4.2.2 Parsning

4.2.3 Evaluering

4.3 Klasser och dess relationer

Vi använder oss av flera klasser för evalueringen av språket, alla klasser har minst en funktion gemensamt, evaluate. Evaluate funktionerna evaluerar de uttryck som finns sparat i klasserna.

4.3.1 Program

Klassen Program är starten för hela programmet och innehåller alla , den har en funktion, 'run', som går igenom alla statements i programmet och evaluerar dom.

4.3.2 Scope

Klassen Scope håller koll på vilket scope programmet är i. Exempelvis, om programmets nuvarande scope är globalt så kommer scope-level vara 0, är vi i en funktion så kommer scope-level vara 1.

4.3.3 Variable

Klassen ”Variable”har bara ett attribut, variabelns namn. Själva värdet på variabeln sparas i den globala variabeln ”variables”, detta gör att klassen Variable är beroende av klassen Scope”eftersom i den globala variabeln ”variablessparas värden relativt till vilken scope-level som är aktiv.

4.3.4 Expression

Klassen Expression innehåller tre attribut:
- left_expr”
- right_expr”
- äritm_op”

left_expröch right_expr”är en instance av någon av dessa typer:
- ”Variable”
- FunctionCall”
- StringConcatenation”
- ÄrrayCall”
- ”Integer”
- String”

Klassens funktion ”evaluate”kallar på left_expr och right_expr's evaluate metod och kör eval på det.

4.3.5 Condition

Klassen Condition innehåller tre attribut:
- left_expr”
- right_expr”
- öperator”

Condition har en direkt relation till Expression eftersom left_expr och right_expr innehåller instancer av klassen Expression.

4.3.6 Assignment

Klassen Assignment innehåller tre attribut:
- ”var”
- ässign_op”
- ”expression”

”var”kan antingen vara av typen ”Variable”eller ÄrrayCall”, och expression är av typen ”Expression”.

4.3.7 AssignmentIncrement

Klassen Assignment innehåller två attribut:
- ”var”
- ”increment_op”

Denna klassens ”evaluatefunktion skapar en ny Assignment och kör evaluate på den, enligt såhär:

```
if @increment_op.eql?('++')
  Assignment.new(@var, '+=', Expression.new(1)).evaluate
else
  Assignment.new(@var, '-=', Expression.new(1)).evaluate
end
```

4.3.8 StringConcatenation

Klassen StringConcatenation innehåller två attribut:
- ”expr1”
- ”expr2”

Denna klassen är direkt relaterad till Expression (vilket nästan alla klasser är), klassens evaluate funktion kör evaluate på expr1 och expr2. Denna klassen är gjord för att sätta ihop två strängar, exempel:

```
a = "string " . "concatenation";
```

4.3.9 Output

Denna klassen har bara ett attribut, ”expr”. Denna klassen är till för att skriva ut ett expression/variabel, exempel:

```
a = "hej";
output(a);
output("hellu " . "hej");
output(20);
```

```
output(true);
```

4.3.10 Input

Klassen Input innehåller tre attribut:
- ”var”
- ässign_op”
- ”expr”

Evaluate funktionen hämtar inputen från stdinöch sparar det i variabeln som är sparad i attributet ”var”.

4.3.11 Block

Klassen Blockänvänds as många olika klasser. Start/stop, funktioner samt if-satser har referenser till en instans av denna klassen för att kunna evaluera allt i blocket.
Klassen innehåller ett attribut, statements, vilket är av typen Ärray”. Den innehåller alla statement som finns i blocket. Dessa statements evalueras senare i klassens evaluate funktion.

4.3.12 If

Klassen If innehåller två stycken attribut:
- condition
- block

Condition innehåller conditionet i if-satsen och block innehåller blocket i if-satsen, demonstrerat nedanför:

```
if(<CONDITION>) <BLOCK>
```

4.3.13 ConditionStatement

ConditionStatement innehåller all information om en if-sats. Den innehåller själva if-satsen, alla elseif-satser samt else-satsen. Klassens evaluate funktion evaluerar alla if-satser som finns sparade. If-satserna som finns sparade i denna klassen är alla instanser av klassen ”If”.

4.3.14 Restart

Denna klassen har inga sparade attribut och det enda som evaluate funktionen gör är att skicka vidare strängen restart”. Klassen används av Start/stop klassen. Om Start/stop klassen innehåller en instans av denna klassen, och denna klassens evaluate funktion kallas kommer Start/stop blocket startas om.

4.3.15 Start

Start klassen har ett attribut, statements, som innehåller alla statement som finns i Start/stop blocket. Klassens evaluate funktion går igenom alla statements och kör dom, om statementet innehåller restartså kommer alla statements att köras om från början.

4.3.16 Return

Return innehåller ett attribut, expression, vilket är av typen ”Expression”. Evaluate funktionen returnerar en array enligt följande:

```
["return", @expression.evaluate]
```

4.3.17 Function

Klassen Function”innehåller all information som finns om en funktion. Attributen som ingår i denna klassen är:
- name
- block
- parameters

block”är av typen Blocköch parameters är en eller flera assignments och expressions.

4.3.18 FunctionCall

Detta är klassen som används när man kallar på en funktion (myFunc()”), den kallar på funktionen och sätter alla parametrar till rätt värde.

4.3.19 Loop

Denna klassen används till följande loopar:

```
loop ( <block> )
loop ( <expression> )
loop ( <expression> , <var> )
loop ( <expression> , <var> , <assignment> )
```

Den innehåller följande attribut:
- block
- count
- varname
- assignment

Block är av typen Blocköch innehåller alla statements mellan måsvingarna. Count är hur många gånger som loopen har körts och varname är den assignade variabeln i <var>, som får värdet av assignment.

4.3.20 LoopWhile

Denna klassen används av följande loop:

```
loop ( <condition> )
```

Klassen har två parametrar, block och condition.
Block fungerar precis som i klassen Loop”, och condition är av typen ”Condition”. Loopen kommer att köras sålänge det conditionet är sant.

4.3.21 LoopFor

Denna klassen används för följande loop:

```
loop ( <assignment>, <condition>, <assignment> )
```

Det första <assignment> är ursprungsvärdet för variabeln som man skickar in. Loopen kommer att köras sålänge <condition> är sant och efter varje iteration så kommer variabeln få värdet av det andra <assignment>.

4.3.22 LoopArray

Denna klassen används till följande loop:

```
loop(<array>, <var>)
```

Detta är en loop som går igenom en array enligt följande:

```
loop([1,2,3], i)
{
    // Första iterationen kommer i == 1
    // Andra iterationen kommer i == 2
    // Trejde iterationen kommer i == 3
}
```

4.3.23 ArrayCall

Denna klassen används när man använder en array, till exempel när man försöker hämta ett värde från en array. Klassen har två attribut, ”varöch ”index_expr”. ”index_expr”är det indexet från array:en programmet försöker nå och ”var”är namnet på arrayen.

Välkommen tillbaks!!!

4.3.24 Null

Det enda denna klassen gör är att returna nil”. Detta eftersom i termer så betyder null samma sak som ruby’s nil”.

4.4 Val av representation av syntaxträd

4.5 Algoritmer

4.6 Kodstandard

4.7 Användning och installation

5 Erfarenheter och reflektion

5.1 Vad gick lättare och vad var svårare än vi trodde?

Det var betydligt enklare att komma igång än vad vi trodde det skulle vara, att kunna ge en variabel ett värde var enkelt först. Det svåra blev när vi började göra klasser till Variabler,uttryck och villkor. Att få allt att fungera utan att behöva göra massa specialfall för olika evaluations.

Jag tyckte även det var svårt att felsöka rdparse.rb, från början så var det svårt att veta vart i en rule jag skulle skapa en ny instans av objektet man försöker skapa.

5.2 Hur mycket kunde vi följa språkspecifikationen?

Vi följde vår språkspecifikation med några undantag, samt att vi inte hade tid att göra allt vi hade planerat. Vi hade aldrig tid att implementera klasser i termer, detta eftersom vi underskattade hur lång tid det skulle ta att implementera allt annat samt att leta och fixa buggar.

Vi valde att inte implementera följande:

```
if(x == 5 | 4 | 10 | null)
```

Detta skulle betyda att x kan antingen vara 5, 4, 10 eller null för att if-satsen ska evalueras till true. Nu skrivs if-satser såhär istället:

```
if(x == 5 || x == 4 || x == 10 || x == null)
```

Vi implementerade aldrig heller:

```
loop ( <expression> , <assignment>, <assignment> ) <block>
```

Detta kändes onödigt då vi redan hade loopar som mer eller mindre gjorde exakt samma sak.

I termer så valde vi att inte tillåta deklarationer av variabler. För att deklarera en variabel måste den samtidigt assignas ett värde enligt följande:

```
a = "hej"; // Detta är OK!
b; // inte OK!
```

Detta är något som vi skulle ha ändrat på om vi fick göra om språket igen då vi nu tycker det är ologiskt att inte kunna deklarera en variabel utan att assigna den till ett värde.

Yes, jag löser det (y) :D låtstå exde ok Det enda vi inte hade tid var klasser. Det andra bara valde vi att inte implementera. Vi kanske också kan nämna att vi inte hade tid att tilldela variabler villkorsuttryck