LINKÖPINGS UNIVERSITET

TDP019 - Datorspråk datormiljön

RaukLang

Författare

Erik Westerlund, eriwe394@student.liu.se
Jesper Olofsson, jesol567@student.liu.se

Höstterminen 2017
Version 1.2

2017-05-14

INSTITUTIONEN FÖR DATAVETENSKAP

Innehåll

1 Revisionshistorik **2**

2 Inledning **2**
2.1 Bakgrund . . . 2

3 Användarhandledning **3**
3.1 För att köra RaukLang . . . 3
3.2 Kommentarer . . . 3
3.3 Datatyper . . . 3
3.4 Matematiska uttryck . . . 3
3.5 Villkorssatser . . . 4
3.6 Jämförelseoperatorer . . . 5
3.7 Repetitionssatser . . . 5
3.7.1 For-loop . . . 6
3.7.2 While-loop . . . 6
3.8 Print . . . 6
3.9 Deklaration . . . 7
3.10 Funktioner . . . 7
3.10.1 Deklarera funktioner . . . 7
3.10.2 Använda funktioner . . . 7

4 Systemdokumentation **8**
4.1 Grammatik . . . 8
4.2 Kodstandard . . . 10
4.3 När RaukLang körs . . . 10
4.3.1 Tokens . . . 10
4.3.2 Regler . . . 11
4.3.3 Klasser . . . 11

5 Erfarenheter och reflektioner **12**

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Skapade dokument	170514
1.1	La till användarhandledning	170515
1.2	Blev klara med första utkastet	170516

2 Inledning

I denna dokumentation kommer vi att gå igenom hur programspråket RaukLang är uppbyggt och hur det används både syntaxmässigt men även grundläggande som att köra sin egen kod. Vi kommer även att gå igenom andra saker så som användningsområde och syfte med språket.

2.1 Bakgrund

Språket utvecklades för kursen TDP019 Projekt: Datorspråk av studenterna Erik Westerlund och Jesper Olofsson, som går första året, termin två på programmet Innovativ Programmering. RaukLang är inspirerat av bland annat Python samt C++. Det är utvecklat för att vara ett relativt simpelt språk just för att vi själva ville ha så bra koll som möjligt på exakt vad vi utvecklat och få med sig så bra förståelse över detta som möjligt. RaukLang är även utvecklat för att ha en bra läsbarhet.

3 Användarhandledning

3.1 För att köra RaukLang

Det man behöver för att köra sin kod i RaukLang är Ruby 1.9, samt de tre ruby-filerna rdparse.rb, parse.rb och RaukLang.rb. Sedan lägger man sin kod i en fil i samma katalog som dessa tre filer. Filändelsen spelar ingen roll. Sedan kör man koden genom att skriva 'ruby parse.rb FILNAMN' i en terminal. Det går också att köra kod interaktivt. För detta skriver man endast 'ruby parse.rb'

3.2 Kommentarer

Är det så att man vill skriva något som ska ignoreras och inte räknas som kod så skriver man $ framför det man vill skriva så kommer det som är efter på raden att ignoreras. Detta kan användas för att kommentera koden men också för att kommentera bort en kodrad istället för att ta bort koden.

```
$Detta är ett exempel på en kommentar och kommer ignoreras
```

3.3 Datatyper

I RaukLang finns det 4 olika datatyper. Då språket är otypat behövs det inte deklareras och man kan ändra på datatypen i en redan satt variabel. Nedan står det hur de olika datatyperna ser ut i våran syntax.

```
Int = 4;
Float = 4.5;
String = "RaukLang";
Bool = true;
```

3.4 Matematiska uttryck

I vårat språk så finns det 4 olika matematiska uttryck. Vårat språk prioriterar först multiplikation och division sedan addition och subtraktion.

```
mult = 5 * 5;
div = 4 / 2;
add = 2 + 2;
sub = 4 - 2;
```

I vårat språk så finns tyvärr inte stödet för några parantesuttryck utan det är något som får göras separat. Det går självklart att skriva fler än endast en av dessa matematiska uttryck på samma rad, exempelvis som needan.

```
print(5+5*3);
```

Detta kommer alltså att skriva ut talet 20 eftersom multiplikation prioriteras högre än addition.

3.5 Villkorssatser

Precis som i de flesta andra språk så har vi så kallade if-satser. Ifall det som står som villkor stämmer så kommer det som står innanför if-satsen att utföras och om inte så kommer programet fortsätta till antingen en else-if-sats som fungerar på exakt samma sätt som en if-sats. Om ingen av If, else-if-satserna stämmer så kommer det som står i ett så kallat else uttryck att utföras, om det inte finns ett else uttryck och ingen av villkorssatserna stämmer som kommer programmet att fortsätta vidare utan att göra något av det som står innanför villkorssatserna.

```
if( 1 != 0):
    print("sant");
else_if;
```

I det första exmplet som ses precis över så kommer endast sant att skrivas ut om det är så att 1 inte är lika med 0.

```
if( 1 == 0):
    print("sant");
else:
    print("falskt");
end_if;
```

I exemplet ovan så kommer det som står innanför paranetserna vid if-satsen att evalueras. Eftersom det inte stämmer så kommer programmet att gåvidare till else-satsen och skriva ut det som är skriver inom den.

```
if( 1 == 0):
    print("sant");
else_if( 1 == 1):
    print("sant");
else:
    print("falskt");
end_if;
```

I det sista exemplet så kommer exakt samma sak som hände innan att hända, förutom att innan programet går vidare till else-satsen så kommer sanningsuttrycker som står innanför else if satsen att evalueras och om det stämmer så kommer koden inom satsblocket att utföras och om det inte skulle stämma skulle programet fortsätta till else-satsen.

3.6 Jämförelseoperatorer

För att kunna se om villkorssatser är sanna, och därmed ska utföras, behövs det jämförelseoperatorer som jämför två olika utryck. dessa uttryck kan vara både variabler och matematiska utryck, och de kan även blandas.

```
x == y;
```

I exemplet ovan behöver variabeln x ha samma värde som variablen y för att vara sann.

```
x != y
```

I exemplet ovan behöver variabeln x ha ett annat värde än vad variablen y har för värde för att vara sann.

```
x > y
```

I exemplet ovan behöver variabeln x ha ett värde som är större än vad variablen y har för värde för att vara sann.

```
x < y
```

I exemplet ovan behöver variabeln x ha ett värde som är mindre än vad variablen y har för värde för att vara sann.

```
x >= y
```

I exemplet ovan behöver variabeln x ha ett värde som är större eller lika stort som värde på variablen y för att vara sann.

```
x <= y
```

I exemplet ovan behöver variabeln x ha ett värde som är mindre eller lika stort som värde på variablen y för att vara sann.

```
x < y \&\& y < 10 $En så kallad 'och' operator
```

Om det är så att man vill att det ska vara fler än ett vilkor som ska uppfyllas så kan man använda sig av &&. När man använder sig av && så krävs det att det som står till höger och vänster om && båda stämmer för att opperationen ska utföras

```
x < y || y < 10 $Detta kallas för en 'eller' operator
```

Nästan som 'och' operatorn. Den stora skillnaden är att endast ett av jemförelseuttrycken behöver stämma istället för att både det som står till höger och vänster måste stämma. Sålänge något av uttrycken stämmer kommer opperationen att utföras.

3.7 Repetitionssatser

I RaukLang så finns det 2 olika sorters loopar, en så kallad for-loop och en så kallad while-loop.

3.7.1 For-loop

För att skriva en så kallad for-loop i RaukLang så ser syntaxen ut på följande sätt

```
for(i = 0; i < 10; i = i + 1):
    print(i); $kommer skriva ut 0 till och med 9
end_for;
```

Först så deklareras en variabel som kommer användas till att veta när loopen ska avslutas. Detta värde kommer ändras för varje loop. I detta exempel kommer det värdet adderas med 1 varje loop, tack vare í = i + 1ÄAllt innanför loopen kommer utföras en gång varje gång loopen går ett varv, och loopen kommer avslutas när villkorsoperatorn blir falsk. Loopen i exemplet kommer alltså köras 10 gånger.

3.7.2 While-loop

En While-loop är för den som inte vet en loop som kommer att utföra koden innanför loopen enda tills sanningssatsen inte längre är sann.

```
i = 5;
while(i < 10):
    print(i);
    i = i + 1;
end_while;
```

Det som kommer hända i kodblocket precis åvan är alltså att variabeln i kommer att skrivas ut en gång sedan adderas med 1 och detta kommer hände enda tills i inte längre är mindre än 10.

3.8 Print

Om något ska skrivas ut i terminalen så skrivs följande i programmet.

```
print("utskrift");
```

Detta kommer att skriva ut texten ”utskrift”i terminalen.

```
print(a);
```

Om a är en satt variabel kommer dess värde att skrivas ut i terminalen.

```
print(a + "utskrift");
```

Detta kommer att skriva ut både a:s värde och texten ”utskriftpå samma rad.

3.9 Deklaration

För att deklarera en variabel i RaukLang behövs följande skrivas:

```
a = 3;
a = 3 + 4;
a = "text";
```

Då kommer det värde man skriver efter ā tt sparas ner i den variabeln innan ¯. Som synes i exemplet går det också att använda matematik i deklarationen, här kommer siffran 7 sparas ner.

3.10 Funktioner

3.10.1 Deklarera funktioner

När man ska deklarera en funktion skrivs följande:

```
func foo(a, b):
    print(a + b);
end_func;
```

I ovanstående exempel är foo funktionsnamnet som används för att använda funktionen och a samt b är invärden som ska användas i funktionen. Sedan kan såklart print(a + b); bytas ut mot det man vill att funktionen ska göras.

```
func bar();
    print("RaukLang");
end_func;
```

Det är inte ett måste att skicka in invärden. Isåfall skriver man som ovan.

3.10.2 Använda funktioner

Om ovanstående exempel på en funktion ska användas gör man följande:

```
foo(c, d);
$alternativt
foo(1, 2);
$alternativt
bar();
```

För första exemplet krävs det att variablerna c och d är deklarerade tidigare i koden. Observera att det är viktigt att skicka in lika många invärden som är definierat i funktionsdeklarationen.

4 Systemdokumentation

4.1 Grammatik

```
<begin>
::=  <stmt_list>

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

<stmt>
::= <for_stmt>
| <while_stmt>
| <assign>
| <print_stmt>
| <if_stmt>
| <return_stmt>
| <func_def>
| <func_call>

<for_stmt>
::= 'for' '(' <name> '=' <add_expr> ';' <bool_expr> ';' <name> '=' <add_expr> ')' ':' <stmt

<while_stmt>
::=  'while' '(' <bool_expr> ')' ':' <stmt_list> 'end_while;'

<if_stmt>
::=  'if' '(' <bool_expr> ')' ';' <stmt_list> 'end_if;'

<print_stmt>
::=  'print' '(' <add_expr> ')' ';'

<return_stmt>
::=  'return' '(' <add_expr> ')' ';'

<func_def>
::= 'func' <name> '(' <call_list> ')' ';' <stmt_list> 'end_func;'

<func_call>
::=  <name> '(' <call_list> ')' ';'
| <name> '('')' ';'
```

```
<call_list>
::= <call_list> ',' <add_expr>
| <add_expr>

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

<bool_expr>
::= <bool_expr> <bool_or> <bool_expr>
| <add_expr> <comp_op> <add_expr>
| <bool>

<bool_or>
::= '||'
| <bool_and>

bool_and>
::= '&&'

<bool>
::= true
| false

<add_expr>
::= <add_expr> <add_op> <mult_expr>
| <mult_expr>

<add_op>
::= '+'
| '-'

<mult_expr>
::= <mult_expr> <mult_op> <factor>
| <factor>

<mult_op>
::= '*'
```

```
| '/'

<factor>
::= <int>
| <string>
| <func_call>
| <float>
| <name>

<int>
::= [0-9]+

<string>
::= "([^"]+)"

<float>
::= [0-9]+.[0-9]+

<name>
::= [A-Za-z]+
```

4.2 Kodstandard

Som man kan se i föregående avsnitt så skriver man sin kod på engelska och det mesta av namnen på identifierarna kan även hittas i andra populära språk. Detta är för att vi inte ville att det skulle bli förvirrande för de som programmerat i andra språk att lära sig helt nya namn på funktionerna i språket. Det går även att se att det används många semikolon och end_xx. Detta gjorde vi för att vi ville ha en bra läsbarhet, så det går enkelt att se när till exempel en loop slutar.

4.3 När RaukLang körs

4.3.1 Tokens

När språket ska tolka kod använder det först 'parse.rb' och lägger alla karaktärer i såkallade tokens och sparar ner de flesta av dessa. RaukLangs tokens ser ut som följande:

```
token(/\s+/)        #Mellanslag
token(/\t+/)        #Tab
token(/\d+/)        #Integer
token(/\$.*/)       #Kommentarer
token(/[a-zA-Z_]+/) #Karaktärer
```

```
token(/"[^"]+"/)    #Strängar

#operators
token(/</)
token(/>/)
token(/&&/)
token(/\|\|/)
token(/\<\=/)
token(/\>\=/)
token(/\!\=/)
token(/\=\=/)

token(/./)          #allting annat
```

RaukLang kommer inte spara ner mellanslag, tabbar samt kommentarer då dessa inte ska köras som kod. Allting annat kommer sparas som tokens.

4.3.2 Regler

När alla tokens är sparade kommer RaukLang matcha dessa mot olika regler som bestämmer syntaxen. Om det till exempel står:

```
print("hej");
```

så kommer språket först hitta ordet 'print', sedan kollar den om det står '(' efter, och så vidare. När språket har matchat mot en hel regel så vet den att den senare ska köras som kod. Då skapar den ett objekt av den klassen som regeln tillhör och sparar de värden den har fått i objektet.

4.3.3 Klasser

Alla regler i RaukLang har en egen klass, borträknat datatyperna. När all kod har matchat mot en regel och skapat klassobjekt av dessa går språket vidare till filen 'RaukLang.rb', där den kör funktionen eval i klassen Final, där den sedan kommer skicka vidare varje objekt mot dess egna eval-funktion, där det kodat vad varje objekt ska göra.

5 Erfarenheter och reflektioner

Direkt när vi startade hade vi svårt att komma på idéer för vad vi ville att vårt språk skulle gå ut på. När vi väl kom på att vi ville ha ett lättläsligt språk så kom vi igång med grammatiken väldigt bra. I början när vi skulle implementera språket gick det också bra, även om vi märkte ganska snabbt att vi var tvungna att ändra vår grammatik på några få ställen. Till exempel var vi tvungna att skriva om jämförelseregeln då vår bool inte fungerade något bra. Då vi också hade jobbat med regler på liknande sätt i kursen TDP007, så gick det ganska bra. Vi kom dock igång med implementeringen av klasser ganska sent, då vi hade en annan tung kurs samtidigt som var tvungen att bli klar. Det med att skriva klasser och eval-funktioner kände vi också var det svåra i och med att vi inte hade jobbat med det tidigare.

Nu när vi är klara med vårt språk så kommer det vi tar med oss mest vara förståelsen över hur programmeringsspråk fungerar när de är byggda på andra språk.