LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

EasyCode

Denis Ivan Blazevic, denbl369@student.liu.se
Jonas Malmlöf, jonma993@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2017
Version 1.2

2017-06-01

Sammanfattning

Programmeringsspråket 'EasyCode' är ett resultat från kursen TDP019 på programmet Innovativ Programmering vid Linköpings universitet. Projektets uppgift var att implementera ett eget programmeringsspråk med hjälp av Ruby och en parser skriven i Ruby.

'EasyCode' är skapat för nybörjare som aldrig har programmerat förr. Språket är menat att hjälpa nybörjare att programmera genom att lära dem syntax som finns i de flesta språk, fast på ett enklare sätt. Inspirationen kommer från Python, Ruby samt C++. Språket är väldigt 'verbose' när man använder sig utav det, dvs. koden skrivs som vanliga meningar nästan.

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första versionen av dokumentationen.	170515
1.1	Korrigerad version av dokumentationen.	170526
1.2	Korrigerad version av dokumentationen.	170601

Innehåll

1 Inledning

Detta projekt har genomförts av två studenter på Innovativ Programming, år 1, vid Linköpings universitet. Rapporten består av fyra delar: inledning till projektet, användarhandledning, systemdokumentation samt reflektion.

1.1 Syfte

Syftet med projektet är att lära sig hur ett programspråk är uppbyggt genom att skapa ett eget språk med olika verktyg som finns färdiga men även som man måste bygga upp själv. Detta ska ge en god kunskap om programspråks utveckling.

1.2 Idé

Vår idé till detta projekt var att kombinera de olika delar som vi gillar hos andra programmeringsspråk: Python, Ruby och även C++. Vi ville ha Ruby och Pythons enkla syntax, och Pythons sätt att köra funktioner på, men även C++ koduppbyggnad. Vi valde att inte köra på ett typat språk, men datatyper används i bakgrunden när språket körs och läser in kod.

1.3 Målgrupp

'EasyCode' är ett enkelt språk som riktar sig mot nybörjare som aldrig har programmerat innan. Det finns inga datatyper som de måste oroa sig över, och syntaxen kan man lära sig väldigt snabbt. Språket är även menat för de nybörjare som sedan vill tackla ett större typat språk såsom C++ eller Java.

2 Användarhandledning

2.1 Installation

Först och främst installera Ruby (om det redan inte är gjort).

Om ett Linux system används kan man installera Ruby genom pakethanteraren som används just till din Linux distribution.

Om ett Windows system används, räcker det med att navigera till `https://rubyinstaller.org/` och ladda ner installationen för Ruby.

2.2 Hämtning utav språket

För att kunna använda språket måste det först laddas ner.

I en webbläsare, navigera till `https://gitlab.ida.liu.se/denbl369/TDP019/repository/archive.zip?ref=master`. Detta kommer ladda ner en '.zip' fil som ska packas upp till en mapp som man själv väljer.

2.3 Köra kodexempel

För att lära sig hur språkets syntax fungerar rekommenderas det att undersöka de kodexempel som finns i mappen '**src/tests**. För att köra dessa exempel, navigera till testmappen i din terminal. Skriv sedan:
`ruby ../EasyCode.rb -f 'filnamn'`
där 'filnamn' ändras till filnamnet på exemplet som ska köras.

2.4 Skapa egna program

För att skapa egna program som ska köras måste koden först skrivas ner i en textfil som man sedan kör som man gör med kodexemplen.

2.4.1 Språkstruktur

Språket har ej start- och stoppblock. Koden körs från toppen av koden till botten. Det vill säga att variabler och funktioner måste ha deklarerats innan de används!
Detta kommer inte fungera:

```
print wrong_variable
"Error: wrong_variable not assigned."
```

Medan detta fungerar:

```
set right_variable to 'hej'
print right_variable
->'hej'
```

2.4.2 Räckvidd av deklarerade satser

Räckvidd, även kallat för 'scopes', används för att hantera satsernas räckvidd. En variabel som är deklarerad i en while-loop kommer enbart att finnas i den loopen, och ingen annan sats kan komma åt den variabeln såvida den inte returneras.

Scope skapas genom de satser som behöver det. I varje scope finns det listor på funktioner som finns i detta scope, variabler i detta scope samt vilken parent scope man har.

2.4.3 Kommentarer

Kommentar i språket skrivs med # och avslutas med en ny rad.

```
# Kommentarer klarar endast av en rad.
# Kommentarer som dessa,
fungerar ej! <- ERROR
```

2.4.4 Datatyper

I 'EasyCode' finns det datatyper, men inget som programmeraren måste tänka på förutom att särskilja på nummer och strängar. Nedan följer exempel på hur man deklarerar en nummervariabel, och strängvariabel.

```
set <variable_name> to <number>
set <variable_name> to '<string>'
```

Observera att strängar måste börja och sluta med ', medan nummer inte får göra det.

2.4.5 Aritmetik

Enkel aritmetik går att göra, men även lite mer avancerade uträkningar.

```
set a to 1 + 5
print a # prints 6

set c to equation((3+2)*5);
print c; # print 25
```

2.4.6 Funktioner

I 'EasyCode' finns det funktioner som kan skicka tillbaka värden. Dessa funktioner är dock endast testade med enkla returvärden då språket ska vara så nybörjarvänligt som möjligt. Exempel på funktion:

```
create function print_if_greater_than_two(variable_to_print)
{
        if variable_to_print greater than 2
        {
                print variable_to_print;
        };
};

set a to 2;

call function print_if_greater_than_two(a); #dont print anything

increase a;
call function print_if_greater_than_two(a); #prints a
```

Tekniskt sett kan man ändra variablen 'a' från funktionen, men detta är dålig praxis och visas inte.

2.4.7 Operatorer

För aritmetik och andra operationer, används två olika kategorier.

- Aritmetiska; +, -, *, /
- Logiska operatorer: equals, not equals, greater than, lesser than, greater or equal to, lesser or equal to.

2.4.8 Utmatning

Inmatning finns ej i nuvarande version. Utmatning finns däremot, och kan skriva ut variabel data samt nummer och strängar.

```
set a to 1;
print a; # prints 1

set a to 'test';
print a; # prints "test"
```

2.4.9 Villkorssatser

Språket har enkla If-satser som kan användas men det finns inga Elseif-satser. Tanken är att det ska vara så enkelt som möjligt, men Elseif-satser kan implementeras i framtiden om det anses behövas.

```
if 3 equals 5
{
        print 'something icky here';
};

if 3 lesser than 5
{
        print 'true';
};
```

2.4.10 Loops

Vi har två stycken loops: while-loops och for-loops. While-loops ser exakt ut som traditionella while-loops. Däremot är for-loopen lite annorlunda. I for-loopen ökas variabeln alltid med ett - detta betyder att man inte kan välja hur mycket for-loopen ska stega eller om den ska stega negativt.

```
#while loop
s = 0;

while s not equals 10
{
        print s;
        increase s;
};

#for loop
set ftest to 0;

for ftest not equals 10
{
        print 'loop';
};
```

2.4.11 Return

'EasyCode' har retur-satser som ger tillbaka värden.

```
set a to 10;
return a; #prints 10

create function testt(abc)
{
return 5;
};

testt = call function testt(ab);

print testt; #5
```

3 Systemdokumentation

I detta kapitel kommer en förklaring på hur 'EasyCode' är uppbyggt.

3.1 Lexikalisk analys

Språket använder sig utav en parser som heter 'RDParser', som är gjord för kursen TDP007 och TDP019. Den fungerar genom att först göra en så kallad lexikalisk analys där tokens, som är en sekvens utav tecken (som är beskrivna med ett reguljärt uttryck), skapas när koden läses igenom av parsern. När denna analys är färdig, skickas en lista tillbaka med alla tokens.

3.2 Parsing

Parsen tar listan som har fåtts från analysen och matchar alla tokens till de grammatikregler som finns beskrivna i koden. När en token passar in på en specifik regel, körs specifik kod för just den regeln och skapar noder som används vid senare tillfällen.

3.3 Parseträd och dess noder

Ett parseträd byggs upp av noderna som har skapats vid parsing av koden. Dessa noder är olika klassobjekt som innehåller all logik som ska köras genom klassfunktionen 'evaluate'. Sedan när programkoden väl körs, har parsern skapat ett stort parseträd som man följer från toppen ner till vänster. För varje nod som har körts hoppar programmet upp i trädet tills man hittar en ny nod som inte har körts, och upprepar.

3.4 Räckvidd

Räckvidd, eller s.k 'scopes', används för att kunna veta var de olika noderna befinner sig och vilka andra noder de kan 'prata' med. Till exempel kan inte en funktion fråga efter en variabel som är skapad i en annan funktion, såvida man inte har returnerat variabeln i fråga.

3.5 Kodstandard

I 'EasyCode' har vi enbart en enda regel för kodstandarden, vilket är att alla satser ska avslutas med '**;**' för att det gör koden mer läsbar samt enklare parsad. Annars vill vi att man ska följa 'snake_case' och vara så 'verbose' (mångordigt) som möjligt med sin kod.

4 Grammatik

4.1 BNF

```
Grammatikregler beskrivs nedan.
Teckenförklarning:
| "eller"
+ "följt av"
[] "måste inte finnas"


<program> ::= <statement_list>
<statement_list> ::= <statement> + <statement_terminator> [+ <statement_list>]
<statement_terminator> ::= ';'
<statement> ::= <function_def> | <print> | <if_statement> | <for_statement>
                            | <while_statement> | <return> | <expr>
<expr> ::= <numeric> | <equation> | <assignment> | <comparison> | <expr_call> | <arithmetics>
                  | <identifer> | <print> | <string>
<equation> ::= /(equation\([^'].+\))/
<expr_call> ::= 'call function' + <identifer> + '(' [+<param_list>] + ')'
<return> ::= 'return' + <numeric> | <identifer>
<print> ::= 'print' + <identifer> | <string> | <numeric>
<if_statement> ::= 'if' + <expr> + <condition_body>
<for_statement> ::= 'for' + <expr> + <condition_body>
<while_statement> ::= 'while' + <expr> + <condition_body>
<condition_body> ::= '{' + <statement_list> + '}'
<param_call_list> ::= <expr> [+ ',' + <param_call_list>]
<param_list> ::= <identifer> [+ ',' + <param_list>]
<function_def> ::= 'create function' + <identifer> + '(' + [<param_list> +] ')'
                  + '{' + <statement_list> + '}'
<assignment> ::= 'set' + <identifer> + 'to' + <expr> | <identifer> + '=' + <expr>
                | 'increase' + <identifer> | 'decrease' + <identifer>
<identifer> ::= /(?!\B'[^']*)[a-zA-Z]+(?![^']*'\B)/
<comparison> ::= <identifer> + <compairson_operator> + (<bool> | <identifer> | <numeric> | <string>)
<numeric> + <comparison_operator> + <numeric>
<comparison_operator> ::= 'greater or equal to' | 'lesser or equal to' | 'greater than'
                        | 'lesser than' | 'not equals' | 'equals'
<arithmetics> ::= <numeric> [+ <arith_operator> + <numeric>]
<arith_operator> ::= '*' | '/' | '+' | '-'
<numeric> ::= <float> | <integer>
<float> ::= /-?\d+\.\d+/
<integer> ::= /-?\d+/
<string> ::= /'[^']*'/
<bool> ::= 'true' | 'false'
```

5 Reflektion

Vi har lärt oss mycket om språkuppbyggnad under projektets gång, hur språk är uppbyggda i allmänhet och hur språk fungerar 'behind the scenes'. Vi förväntade oss mycket jobb jämför med vad som behövdes för att få en fungerade version. Däremot tog det lång tid att lista ut hur funktioner ska fungera och hur man ska skapa returvärden. Vi fick även idéer och inspiration från tidigare TDP019-projekt.

De första versionerna fungerade bra men det var otroligt ful och hackig kod! Efter att vi fick allt att fungera som vi ville, satt vi och skrev om hela kodbasen till en mycket finare och mindre ful kodbas. Detta tyckte vi var bra gjort av oss då vi kan nu hitta buggar mycket snabbare och bara allmänt veta var vi ska ska titta i koden när ett fel uppstår.

Det svåraste vi implementerade var funktioner! Det tog oerhört mycket tid då språket hade många buggar som inte var fixade, detta gjorde att vi aldrig kunde evaluera funktions koden ordentligt. Felet med funktionerna var att vi skickade runt en array i en array, vilket gjorde att vår kod 'fungerade' men ingen utskrift gjordes då allt hamnade i fel scope.

Vår första version av grammatiken jämfört med vår version i nuläget, är rätt så lika - vilket tyder på att vi har hållit samma tanke genom projektet och inte ändrat för mycket.

Målet med vårt språk var att nybörjare ska kunna sätta igång med programmering lätt, genom att lära sig den oerhört enkla syntaxen (som inspirerades från Ruby, Python samt C++), och sedan direkt flytta över fokus på ett av de tre språken som inspirerade 'EasyCode'. Poängen med detta språk var aldrig att nybörjarna ska använda språket konstant, men att de ska lära sig hur olika satser fungerar och vad man kan göra med så enkla satser som möjligt. Någon dag eller två är lämpligt att lära sig syntaxen felfritt och sedan fortsätta till ett annat språk.