CodEng

Språkdokumentation

Linköpings Universitet
Innovativ Programmering
TDP019 Projekt, Datorspråk

Johannes Fagerberg, johfa464@liu.se

Viktor Björk, vikbj455@liu.se

Datum: 2020/05/10

Inledning **2**
Introduktion 2
Målgrupp 2

Användarhandledning **3**
Förutsättningar 3
Installation 3
Första programmet 3

Syntax samt strukturer **5**
Datatyper 5
Heltal 5
Flyttal 5
Strängar 5
Boolesk 5
Array 6
Konstruktioner 6
Tilldelning 6
Aritmetiska uttryck 7
Operatorer 9
Satser 9
Funktioner 11
Standardbibliotek 13
Iteration 14
Kommentarer 15
Räckvidd 15

Systemdokumentation **16**
Översikt 16
Lexikalisk analys 16
Parsning 17
Evaluering 17
Grammatik 17

Reflektion **21**

Inledning

CodEng är ett programmeringsspråk som är skapat inom kontexten av projektkursen TDP019 som lärs ut den andra terminen på Innovativ Programmering på Linköpings universitet.

Introduktion

Vårt val av inriktning utgick ifrån att inom programmering så finns det många koncept som kan ses som abstrakta för människor som inte har mycket erfarenhet med att koda. Målet med vårt språk är då att göra dessa koncept mer tillgängliga genom huvudsakligen förändrar syntax. Koncepten finns självklart fortfarande kvar men i en mer användarvänlig form och vi har försökt implementera språket med fokus på att det ska vara enkelt att kunna förstå och koda.

Målgrupp

Med det vi nämnt i delen ”Introduktion” så var vårt fokus att skapa ett så användarvänligt språk som möjligt. Därför är våran målgrupp huvudsakligen nybörjare som eventuellt vill börja koda för första gången i ett språk där de kan lära sig koncepten i en bekant miljö, alltså skriftlig engelska. Till exempel så kräver inte språket användning av indentering utan har regler som härstammar från engelska språkets struktur, med att man måste börja kodrader/meningar med stor bokstav och avsluta med punkt.

Förhoppningsvis så kan språket användas som en språngbräda i en persons kod kunskaper och utveckling, där de kan sedan börja använda och förstå mer avancerade språk och begrepp.

Användarhandledning

Förutsättningar

Först och främst så behöver man ha installerat det språk som språket är utvecklat i. I vårat fall är detta **Ruby**. vi använde version 2.5.1 samt operativsystemet Ubuntu 18.04. För att installera Ruby så hänvisar vi till den officiella dokumentationen, https://www.ruby-lang.org/en/documentation/installation/

För att kunna skriva/redigera kod så behöver man en **Textredigerare**. Vi rekommenderar Atom eller Emacs. Dessa kan laddas ned via deras respektive hemsidor.

Installation

För att köra CodEng så öppnar man terminalen för att sedan gå till mappen där filerna finns. För att sedan starta ett program, till exempel programmet “test.ce” så gör man följande,

```
>> ruby run.rb test.ce
```

Första programmet

Ett enkelt exempel är följande där vi vill skriva ut “Hello world” i terminalen. Om vi antar att filnamnet är test.ce så kommer det se ut som följande i test.ce

```
Print Start Sentence: Hello world :End Sentence.
```

Vilket då ska se ut som följande i terminalen,

```
>> ruby run.rb test.ce
(Filen körs och får följande utskrift i terminalen)
>> Hello world
```

Härifrån så kan vi prova att spara “Hello world” i en variabel för att sedan skriva ut den. Anledningen till att göra detta är för att det visar på hur vårt språk kan vara annorlunda gentemot andra språk.

I test.ce,

```
Make x equal to Start Sentence: Hello world :End Sentence.
Print the value of x.
```

Vilket då ska se ut som följande i terminalen,

```
>> ruby run.rb test.ce

>> Hello world
```

Anledningen till att saker ser annorlunda ut i det andra exemplet kommer förklaras i nedanstående avsnitt.

Syntax samt strukturer

För att utföra något i CodEng så krävs det för det första att raden eller meningen i vårat språk börjar med en stor bokstav och slutar med en punkt. Detta betyder att CodEng inte är indentering känsligt men i fall såsom iteration och liknande så rekommenderar vi ändå att utnyttja indentering.

Datatyper

CodEng innehåller följande primitiva datatyper, vilket innebär de grundläggande typerna:

Heltal, Flyttal, Strängar, Boolesk

Det finns även en abstrakt datatyp, som till skillnad från de förra kan ses som en datatyp som inkluderar förutom dess typ ett antal operationer man kan utföra på den Denna datatyp är en **Array.**

Heltal

Denna datatyp gäller alla siffror som är heltal, vare sig dem är positiva eller negativa.

Flyttal

Inkluderar alla tal som har decimaler i sig.

Strängar

Allt som ses som “text”, ord eller liknande innebär att det är en sträng. För att skapa en variabel så måste namnet på variabeln vara en sträng.

Till exempel,

```
Make textexempel equal to 5. # Skapar variabeln textexempel och ger den värdet 5.
Print the value of textexempel. #Skriver ut 5.
```

Boolesk

Representerar två värden, sant och falskt. I CodEng används dessa bland annat som returvärden från operatorer samt för att utvärdera “If-satser”. Variabler kan även sättas till true eller false.

Array

En form av lista. Eftersom att det är en abstrakt datatyp så finns det specifika operatorer som används med denna datatyp. I CodEng så finns “add” och “remove” implementerat som lägger till element samt tar bort dem. Man kan även använda regeln “arrayValue” för att ta fram ett specifikt värde i listan.

Till exempel,

```
Make y equal to List: 5 and 4 and 3 and 2 :End List. #Skapar en variabel y som är av datatyp lista och fyller den med 5, 4 ,3 och 2.

Print the value of y. #Skriver ut y ([5, 4, 3, 2]).


Print the position: 1 in y. #Skriver ut värdet på index 1(4).

Print the result of 5 plus the position: 0 in y. #Skriver ut värdet 5+värdet som är på index 0 i y(5) vilket blir = 10.

Add 6 to y. #Lägger till 6 i y
Print the value of y. #Skriver ut listan i ordningen som värdena lades in ([5, 4, 3, 2, 6]).
Add List: 8 and 9 :End List to y. #Lägger till en lista i y med värdena 8 och 9.
Print the value of y. #Skriver ut y ([5, 4, 3, 2, 6, [8, 9]])
```

Konstruktioner

Konstruktionerna som finns i CodEng är, **Tilldelning**, **Aritmetiska uttryck,Operatorer, Satser, Funktioner, Iteration** (For samt While loopar)

Tilldelning

Den tilldelning som finns implementerad i CodEng är huvudsakligen variabeltilldelning. För att detta ska kunna utföras så krävs det att namnet på variabeln startar med en bokstav(stor eller liten har ingen betydelse) samt att variabeln sätts till ett värde.

Detta värde kan vara allt ifrån en tom lista till ett matematiskt uttryck.

Exempel,

```
Make x equal to 5. #Skapar x till 5.
Make x equal to negative 5. #skapar x till -5
Make x equal to the result of negative 5 minus 2. #Skapar x till -5 -2 => -7
Make x equal to the result of 10.5 minus 2.5. #Skapar x till resultatet 10.5-2.5 => 8.
Make y equal to the result of the value of  x plus 5. #Skapar y till x + 5 => 13.

Create Testfunction with the instructions:

    Return 10.

    stop.

Make x equal to the result of Use Testfunction. # Skapar x till resultatet av funktionen
Testfunction => 10.

Make y equal to the value of x. #Skapar y till värdet i variabeln x => 10.

Make y equal to List: 5 and 2 :End List. #Skapar en variabel y som är av datatyp array och
fyller den med 5 och 2.

Make x equal to user input. #Använder input för att skapa x till ett användar inmatat värde.
```

Det går även att sätta variabler till text(strängar) alternativt listor innehållande strängar.

Som i exemplet så gäller det vid utskrift av värdet som en variabel satts till att man använder “the value of”, annars skriver den ut variabelnamnet. Detta gäller även vid andra tillfällen där man vill använda värdet som är sparat i en variabel.

Aritmetiska uttryck

CodEng utnyttjar multiplikation, division, addition samt subtraktion. Till dessa så används de “vanliga” (?) matematiska reglerna. Parenteser kan man även utnyttja. Man måste antingen spara resultatet i en variabel eller skriva ut det direkt i terminalen. uttrycken kan endast appliceras på datatyperna heltal och flyttal. För att visa vid t.e.x variabeltilldelning att variabeln ska sättas till resultatet av ett matematiskt uttryck så använder man “the result of” innan.

Exempel,

```
Make x equal to the result of 10 plus 5. #Sparar 15 (10 + 5)  i variabeln x.

Make y equal to the result of (5 plus 5) multiplied with 6. #Sparar 60((5+5) *6) i variabeln x.

Make x equal to the result of 10 multiplied with 5 divided with 10. #Sparar 5 (10*5/10) i
variabeln x.

Make y equal to the result of the value of x multiplied with 6 plus (2 multiplied with 5).
#Med antagandet att x = 5 så sparas 40 (5*6+(2*5)) i variabeln y.

Print the result of 1.5 multiplied with 1.5. #Skriver ut 1.5 * 1.5 vilket blir 2.25
```

Det finns även ökning/minskning av variabler. För att dessa ska fungera så måste variablerna redan vara deklarerade.

Exempel,

```
Make x equal to 5. #Sparar 5 i x.

Increment x. #Ökar x med 1.

Decrement x. #Minskar x med 1.
```

Operatorer

Följande operatorer finns i CodEng. Operatorerna som finns implementerade följer boolesk algebra, vilket innebär att de returnerar sanningsvärden. Exempel på dessa finns nedan.

```
Print the result of 2 greater than or equal to 4. # ” >=” vilket returnerar falskt.

Print the result of 5 greater than 4.  # “>” vilket returnerar sant.

Print the result of 2 smaller than or equal to 4.  # “<=” vilket returnerar sant.

Print the result of 2 smaller than 4.  # “<” vilket returnerar sant.

Print the result of 4 equal to  4.  # ” ==” vilket returnerar sant.

Print the result of 4 not equal to 4.  # “!=” vilket returnerar falskt.

Make x equal to 2. #Skapar x med värdet 2.

Print the result of 4 not equal to the value of x and the value of x greater than 3. # “!=”, “and/&&” samt “>” vilket returnerar sant då båda satserna är sanna.
```

Satser

Operatorer används huvudsakligen i satser såsom “If” samt “else” i till exempel Python. Detta utförs vanligtvis när man vill utföra en operator, där baserat på svaret utföra olika uppgifter. I CodEng så har man till en början tre delar. Man börjar med “If” för att sedan skriva in den andra delen vilket är en operator. Till sist så skriver man sina “statements” vilket är den kod som man vill ska utföras om satsen går igenom (alltså returnerar sant). För att visa att man skrivit klart sina statements så skriver man “stop.”

För att sedan lägga till en ytterligare sats så skriver man först “Otherwise” plus de tre delarna nämnt ovan. Till sist så kan man välja att ha en som gäller i princip “allt annat”, alltså alla andra fall. Då skriver man bara “Otherwise” plus sina statements.

Exempel,

```
Make x equal to 5.
If the value of x equals 3 then:
    Print the value of x. Print Start Sentence: equal to 3 :End Sentence.
    stop.

Otherwise If the value of x smaller than 3 then:
    Print the value of x. Print Start Sentence: is smaller than 3 :End Sentence.
    stop.

Otherwise:
    Print the value of x. Print Start Sentence: is greater than 3 :End Sentence.
    stop.
```

När man då ger ett värde till x så kommer utskriften förändras baserat på dess storlek i jämförelse med 3.

Det går även att ha “nästlade” if satser. Exempel på det är följande,

```
Make temp equal to 5.

If temp equal to 5 then:
  Print Start Sentence:We are now in the first if:End Sentence.

  If temp equal to 6 then:
    Print wrong.
    stop.
  Otherwise:
    Print Start Sentence:We are now in the nestled else:End Sentence.
  stop.
stop.
Otherwise:
  Print hej.
stop.
```

Funktioner

Funktioner i CodEng har tre block/delar: namnet av funktionen, en valfri parameterlista, och blocket med instruktioner som funktionen ska utföra. Parameterlistan består av ett antal namngivna men inte typade parametrar. Funktionens namn får bestå av stora eller små bokstäver, men får endast vara ett ord långt. Blocket med instruktioner får bestå av alla språkets konstruktioner. CodEng stödjer inte nästlade funktioner.

Funktionen används genom att skriva:

```
Use <funcname>.
# Om en parameterlista var deklarerad till funktionen används istället:
Use <funcname> with the parameters: <parameters>
```

Exempel på hur det kan användas,

```
Create Testfunction with the instructions:
    Make y equal to 5.
    Make x equal to 4.
    Make z equal to the result of the value of y plus the value of x.
    Print the value of z.
    stop.
## Skapar funktionen Testfunction.
Use Testfunction. #Kallar på funktionen
```

```
#Resultat: y = 5, x = 4

# z = y + x

# => z = 5 +4.

# Skriver ut 9.

##################

Create Testfunctiontwo with the parameters: x and y with the instructions:

  Make z equal to the result of the value of x plus the value of y.

  Print the value of z.

  stop.

#Skapar funktionen  Testfunctiontwo.

Use  Testfunctiontwo with the parameters: 2 and 4. # Kallar pÃ¥ funktionen med
parametrar.

# z = x +y

# z = 2 + 4

#Skriver ut 6.
```

Det finns även möjlighet till att använda rekursion.

Exempel på en funktion som använder det är följande,

```
Create Testfunctiontwo with the parameters: x with the instructions:

If the value of x equal to 1 then:

    Print finished.

    stop.

Otherwise:
```

```
    Decrement x.
    Use Testfunctiontwo with the parameters: the value of x.
    stop.
stop.
Use Testfunctiontwo with the parameters: 2.
#Det kommer endast skrivas ut “finished” i terminalen.
```

Standardbibliotek

I CodEng så finns det möjlighet att använda sig av ett standardbibliotek. I vissa programmeringsspråk så görs detta genom att använda sig av funktionsanrop, Till exempel pow(2, 2) som beräknar 2 upphöjt i två. CodEng har två motsvarigheter till dessa funktioner.

Den första är input, som tar in information från användaren via terminalen. Man kan använda den till bland annat variabeltilldelning.

Exempel,

```
Make x equal to user input. # Användaren får mata in ett värde via terminalen.
```

Den andra är “Round” som avrundar flyttal. Man kan välja antalet decimaler samt om den ska avrunda uppåt eller nedåt när talet är “på mitten” (till exempel, 2.5)

Exempel,

```
Print the result of Round 2.59999 with number of decimals equal to: 1.
 # Gör om talet och skriver ut 2.6.
Print the result of Round down 2.55 with number of decimals equal to: 1. # Skriver ut 2.5
Make x equal to 5.55.
Print the result of Round up x with number of decimals equal to: 1. # Skriver ut 5.6
```

Iteration

Om man vill utföra specifik kod flera gånger så kan man använda sig av iteration i form av loopar. I CodEng så finns “for” loopar där man säger först till om hur många iterationer som ska ske. Detta kan göras i form av räckvidd (till exempel 0 till 4) samt baserat på element i en lista.

Den andra iterations möjligheten är “while” loopar, som kommer iterera tills ett visst krav har uppfyllts. Baserat på syntax i andra språk så skulle man kunna anta att det följande exemplet på en while loop är en Do-While loop. Detta är alltså **inte** sant, det är alltså en while loop. Detta betyder att den inte kommer iterera alls om kravet inte är uppfyllt från början.

Exempel,

```
Repeat in range 0 to 2 times:
Print Start Sentence: this is an example :End Sentence.
stop.
#Kommer att skriva ut “this is an example” 2 gånger.

Repeat 2 times:
  Print hej.
  stop.
#IstÃ¤llet fÃ¶r att iterera genom ett omfÃ¥ng sÃ¥ gÃ¥r den frÃ¥n 0 upp till det givna vÃ¤rdet.

Repeat with variable i 5 times:
    Print the value of i.
    stop.
#Denna loop används till att kunna använda värdet på vilken iteration man nuvarande är i inuti
```

```
#själva iterationen. Detta betyder att det kommer skrivas ut 1 2 3 4 5 i terminalen.

Make x equal to 2.

Do:

Make x equal to the result of the value of x plus 1.

As long as: the value of x equal to  6.

### Kommer att öka x med 1 tills x är 6.
```

Kommentarer

Om man vill skriva text i en fil som inte ska exekveras så använder man en “#” följt av det man vill skriva. Man måste ha en “#” på varje rad som ska kommenteras. Exempel på detta kan ses i de tidigare givna exemplen.

Räckvidd

När man kör ett program i CodEng så utgår språket alltid att det finns ett globalt namnrum. Detta rum finns i form av en lista inuti den globala variabeln “var_list” som sparar variabler. Om man sedan skapar en funktion så utökas “var_list” med en till lista där alla **nya** variabler skapas läggs till. Funktionen har alltså ett lokalt namnrum med en egen lista. För att skapa variabler så används en funktion som först kollar i det globala namnrummet, för att sedan skapa den i det lokala om den inte hittar något. Vid situationer där man sedan behöver hitta variabler så kommer de verktyg som utför dessa först och främst se i det lokala namnrummet efter dem. Om den inte kan hitta variabeln där så tittar den i det globala namnrummet.

Sättet detta fungerar på är att varje lista i “var_list” representeras som i alla listor med index. Eftersom det globala namnrummet alltid skapas så kommer det vara indexet 0. När en funktion väl skapas så utökar vi “var_list”, vilket gör att funktionen som nyss skapades kommer ha sin egen lista med sitt eget index. För att veta vilken funktion som ska ha vilket index så har vi en global variabel som heter “highest_scope” som innehåller ett heltal. Detta tal ökar med 1 varje gång vi utökar “var_list” och sänks med 1 varje gång vi avslutar ett funktionsanrop.

Systemdokumentation

Översikt

Den lexikaliska analysen och parsningen hanteras av modulen RDparse. Denna modulen används för att tolka de tokens och regler som vi har skapat i vår implementation. Med dessa skapas sedan ett abstrakt syntaxträd som evalueras vid interpreteringen.

Lexikalisk analys

Den lexikaliska analysen börjar med en sekvens av tecken. Dessa tecken görs sedan om till tokens, som är en sekvens av tecken, baserat på de reglerna vi har använt. Dessa regler består av nyckelord för vårt språk, följt av strängar, och till sist integers och floats. Dessa tokens kommer sedan användas i parsningen.

Parsning

Parsningen börjar efter att den lexikaliska analysen är klar. Då kommer en sekvens av tokens att undersökas, och jämföras med de grammatiska reglerna, som beskrivs nedan i BNF-notation. När en regel har hittats kommer en ny nod att skapas, och med dessa noder kommer ett abstrakt syntaxträd att byggas.

Evaluering

När parsningen är klar ska koden evalueras. Detta görs genom att programmet evaluerar noden som skapades först. Denna noden kommer sedan att se till att dess barn-noder evalueras. Detta upprepas till alla noder i det abstrakta syntaxträdet har evaluerats.

Grammatik

Grammatiken är beskriven med BNF-regler:

<run> ::= <statements>

<statements> ::= <statement><statements>

| <statement>

<statement> ::= <if><end>

| <assign><end>

| <print><end>

| <increment><end>

| <decrement><end>

| <return><end>

| <logical><end>

| <math><end>

| <arrayOps><end>

| <for><end>

| <while><end>

| <createFunc><end>

| <callFunc><end>

| <functions><end>

| <comment>

<comment> ::= /#[\s\S]*?$/

<functions> ::= <roundNumbers>

<roundNumbers> ::= Round up <varname|Float|Integer> with number decimals equal to:
<Integer>

| Round down <varname|Float|Integer> with number decimals equal to:
<Integer>

| Round <varname|Float|Integer> with number decimals equal to:

<Integer>

<createFunc> ::= Create <funcname> with the instructions: <statements> stop

| Create <funcname> with the parameters: <createParam> with the instructions: <statements> stop

<callFunc> ::= Use <funcname>

| Use <funcname> with the paremeters: <callParam>

<funcname> ::= <String>

<createParam> ::= <varname> and <createParam>

| <varname>

<callParam> ::= <parameter> and <callParam>

| <parameter>

<parameter> ::= <values>

<while> ::= Do <statements> As long as: <logical>

<for> ::= Repeat with variable <varname> <value> times: <statements> stop

| Repeat in range <value> to <value> times: <statements> stop

| Repeat <value> times: <statements> stop

<if> ::= If <logical> then: <statements> stop <end> <else>

| If <logical> then: <statements> stop

<else> ::= Otherwise <if>

| Otherwise: <statements> stop

<logical> ::= <and>
| <or>
| <operator>
| <trueFalse>

<and> ::= <operator> and <operator>

<or> ::= <operator> or <operator>

<trueFalse> ::= True
| False

<operator> ::= <greater>
| <smaller>
| <equals>
| <not>

<greater> ::= <values> greater than or equal to <values>
| <values> greater than <values>

<smaller> ::= <values> smaller than or equal to <values>
| <values> smaller than <values>

<equals> ::= <values> equal to <values>

<not> ::= <values> not equal to <values>

<math> ::= <addsub>
| <muldiv>

<addsub> ::= <values> plus <muldiv|values>
| <addsub> plus <expr|values>
| <values> minus <muldiv|values>
| <addsub> minus <expr|values>
| <muldiv>

<muldiv> ::= <muldiv> multiplied with <expr|values>
| <values> multiplied with <expr|values>
| <muldiv> divided with <expr|values>
| <values> divided with <expr|values>
| <expr>

<expr> ::= (<math>)
| the result of <callFunc>

<increment> ::= Increment <varname>

<decrement> ::= Decrement <varname>

<assign> ::= Make <varname> equal to <input>
| Make <varname> equal to <values>

<return> ::= Return <values>

<input> ::= user input

<print> ::= Print <values>

<arrayOps>::= <arrayAdd>
| <arrayRemove>

<arrayAdd> ::= Add <value> to <varname>

```
<arrayRemove> ::= Remove <value> from <varname>
                | Remove the position <Integer> in <varname>
                | Remove the position the value of <varname> in <varname>

<arrayValue> ::= the position the value of <varname> in <array>
               | the position the value of <varname> in <array>
               | the position <Integer> in <array>
               | the position <Integer> in <varname>

<array> ::= List: <arrayValues> :End list
          | List: :End list

<arrayValues> ::= the value of <varname> and <arrayValues>
                | <value> and <arrayValues>
                | the value of <varname>
                | <value>

<values> ::= the result of <callFunction>
           | the result of <functions>
           | the result of <math>
           | the result of <logical>
           | the value of <varname>
           | <value>

<varname> ::= <String>

<value> ::= <trueFalse>
          | <array>
          | <arrayValue>
          | Start Sentence: <String> :End Sentence
          | <String>
          | <Integer>
          | <Float>

<end> ::= .
(OBS, följande används genom Rubys inbyggda datatyper)
<String> ::= /[\w]+/

<Integer> ::= /\d+/

<Float> ::= /[0-9]+\.{1}[0-9]+/
```

Reflektion

Vi hade som mål att skapa ett programmeringsspråk som kunde klara av att bygga vardagliga meningar. Därför valde vi från början att hålla oss ifrån typning, och försökte få mer abstrakta koncept som funktioner och listor att passa in så bra som möjligt.

Vårt första mål med matematiken var att man inte skulle skriva “5 plus 5”, utan istället “Add 5 with 5”, och också använda liknande konstruktioner för de andra aritmetiska operationerna. Vi fick problem när vi försökte implementera dessa regler, eftersom det inte gick att på ett rimligt sätt använda olika operationer i samma utsträckning som vanlig matematisk beräkning. Vi bestämde oss då att använda mer traditionella konstruktioner som “5 plus 5”, vilket gjorde implementeringen av matematiken enklare, utan att ändra målet med språket.

En sak som vi är nöjda med är att vi lyckades implementera och få grunden att fungera. Med grunden så syftar vi på de grundläggande regler för meningsbyggnad, alltså att man börjar meningar med stor bokstav och avslutar med punkt. Med ytterst få undantag så lyckades vi implementera detta och har fått det att fungera “logiskt” i bemärkelse till engelskans språkstruktur. I liknande avseende så var det lite svårt i början att få mer avancerade koncept så som loopar samt funktioner att fungera logiskt sätt. Efter diskussion och tänkande så anser vi att vi lyckades hitta en bra kompromiss på detta problem.

Något som vi inte alls tänkte på när vi skrev vår språkspecifikation var om att implementera ett standardbibliotek, som i många andra språk använder sig av förutbestämda och skrivna funktioner. Vi insåg när vi väl kom igång med projektet att det var värt att lägga tid på detta, huvudsakligen för att kunna visa att den delen av programmeringsspråk kan finnas på ett bra sätt i ett språk som ser ut som vårat. Även om det kanske inte fungerar eller ser perfekt ut i den nuvarande implementationen så var det bra men framförallt roligt att vi fick realisera dessa koncept.

Överlag har projektet varit väldigt lärorikt. Att arbeta med ett eget programmeringsspråk har gett oss mer kunskaper om hur språk fungerar. Till exempel känner vi att det har blivit enklare att förstå varför vissa saker inte tillåts i många av de språk vi använt tidigare.
En annan del av detta är att man har fått utökad förståelse för hur stor ansträngning det krävs för att skapa en sån produkt som vi har gjort.

Att jobba med ett projekt i grupp har varit stundtals utmanande, både arbetsmässigt men även logistiskt sett med tanke på att i princip allt kodande har skett på distans. Framförallt så har det hjälpt att ha någon att prata med när man fastnat med något. När man har haft någon att bara bolla ideér med eller bara allmänt fråga om hjälp så har det underlättat att kunna utföra projektet och göra framsteg.