Linköpings universitet VT 2010
Innovativ Programmering TDP019 Projekt: Datorspråk

pLite

programmering för nybörjare

Jonathan Doherty
jondo466@student.liu.se

Christer Thor
chrth485@student.liu.se

Inledning

Språket pLite är en del av kursen TDP019 Projekt: Datorspråk VT 2010 på Linköpings universitet.

Målet med kursen var att konstruera ett mindre datorspråk i Ruby och implementera verktyg genom en interpretator eller kompilator för det egna språket. Vi skulle diskutera och motivera valen av design i datorspråket med utgångspunkt i teori och egna erfarenheter, samt formulera en teknisk dokumentation av vårt egna datorspråk.

Beskrivning av vårt språk pLite

Detta språk är tänkt som ett nybörjarspråk för gymnasieelever. Namnet pLite tyder på att det är ett simpelt språk för nybörjare i programmering. Tanken är att de kan använda sig av detta språk som "inkörningsport" för mer avancerade programmeringsspråk (som Python, Ruby, C++) i framtiden. Dessutom ska det tvinga dem att skriva kod korrekt enligt vedertagna standarder som många duktiga programmerare tyvärr slarvar med. Exempel på detta är rätt indentering och mellanslag före/efter operatorer.

Vad ni kan förvänta er av pLite

Nu när vi programmerat i Ruby har vi märkt att man kan göra de flesta saker på många olika sätt, vilket ibland blir väldigt förvirrande speciellt om man är nybörjare och det är första gången man stöter på ett nytt språk. För att vårt språk ska vara lätt att lära sig har vi hållt oss till ett sätt att göra saker och det är simpelt men rätt. Detta eftersom vi genom vårt språk vill att de ska lära sig tänka rätt, vilket i sin tur kommer ge dem bättre förståelse för hur man programmerar ”korrekt”. Vi vill också att de ska känna igen sig och ha väldigt lätt att anpassa sig till nya språk med svårare syntax. Samtidigt som de kommer känna igen grundsyntaxen i de flesta språk som if, else, for, funktionsanrop, loopar etcetera. Så att även om de nya språken har en lite annorlunda syntax vet de vad det är och kan lätt anpassa sig till omgivningen utan några större problem.

Användarhandledning

Komma igång!

För att använda pLite krävs det att SciTE och Ruby finns installerat på datorn detta kan göras genom länkarna nedanför. Om man vill kan man

använda någon annan editor som stöder Ruby men instruktionerna här är för SciTE.

SciTE - http://www.scintilla.org/SciTE.html

Ruby - http://www.ruby-lang.org/en/downloads/

För att börja koda måste ni dubbelklicka på SciTE ikonen eller skriva in ***scite &*** i terminalfönstret. Ni kommer nu se ett tomt dokument med namnet ***Untitled***, börja med att klicka på ***File*** och välj sedan ***Save As*** skriv sedan in ett passande namn på dokumentet och avsluta med ***.rb*** t.e.x

test.rb

Om ni vill skapa ett nytt dokument så trycker ni på ***File*** längst upp till höger i SciTE och väljer sedan ***New***.

Tryck sedan på ***File*** och välj ***Open..*** navigera sedan till mappen ***Språk-pLite*** där alla våra filer ligger och välj ***pLite.rb***.

Det kommer nu finnas två dokument

test.rb och ***pLite.rb***

Tryck på ***pLite.rb*** och skriv sedan i det namnet ni valde (i detta exempel ***test.rb)***

T.e.x

```
require 'parser'

#ENTER THE NAME OF YOUR FILE HERE:
file_name = "test.rb"
################################

Parsing_pLite.new().parse_from_file(file_name)
```

Klicka sedan på ***test.rb*** för att börja skriva er kod.

När ni har skrivit en bit av kod ni vill köra(testa) så trycker ni bara på ***pLite.rb*** och trycker sedan på ***F5*** för att se resultatet av kompileringen.

Snabkommandon!

Spara - Ctrl + S
Öppna - Ctrl + O
Ny fil - Ctrl + N
Spara som - Skift + Ctrl + S
Skifta mellan filerna - Ctrl + Tab

Tekniker

Nedan kommer en genomgång av alla tekniker som används i vårt programmeringsspråk pLite för att skriva en fungerande kod.

Variabler

En variabel är en pekare till en plats i minnet där data kan lagras, ersättas, ändras.

En variabel kan liknas vid en låda. Variabelnamnet är namnet du skriver på lådan för att kunna hitta den bland alla andra lådor. Värdet på variabeln är innehållet i lådan. Variabeltypen kan liknas vid olika sorters lådor. Till exempel har man skokartonger för skor och matlådor för mat. på samma sätt har man `int` för heltal (intlådor) och `float` för flyttal/ decimaltal (floatlådor). På samma sätt som du inte får lägga mat i skokartonger kan du inte lägga ett flyttal i en intvariabel!

För att använda en variabel måste man först deklarera den så omgivningen vet att den finns samt ange vilken typ av data den ska innehålla. Sedan måste man tilldela den ett värde som stämmer in på vilken typ den är.

```
int my_variable = 0
```

`int` är alltså typen och `my_variable` namnet. Värdet på variabeln är just nu `0`.

Det finns flera olika typer du kan ge variabeln.

Typ	Värde till typen
`int`	Integer, alltså heltal som `42`.
`float`	Flyttal/decimaltal, till exempel `5.101`
`string`	en sträng av valfria tecken, begränsas med två " eller två '. exmpelvis `"valfri text här emellan citattecknen"` eller `'med apostrofer kan man skriva " mitt i en sträng!'`

`bool` och — Boolean, på svenska boolesk, som är antingen sann eller falsk skrivs `True`, `true`, `False` eller `false`

Kommentarer

Ibland är det bra att skriva förklaringar till koden, men börjar man skriva text direkt i en kodfil försöker datorn köra det som programmeringskod vilket inte är särskilt bra. Därför finns speciella tecken för att kunna kommentera koden. Det finns två varianter på dessa i pLite. Den första skrivs med // följlt av kommentartexten, och gäller från // till nästa rad börjar. Den andra fungerar för fler rader, och börjar med /* och slutar med */. Några exempel på kommentarer:

```
//Det här är en enradskommentar.
int i = 0 //Man kan kommentera på samma rad som kod också.

/*
Den här kommentaren funkar
på flera rader!
*/
```

Aritmetiska uttryck

Aritmetiska (matematiska) uttryck används för att beräkna värden, och följer samma regler som enkel matematik. Några exempel på aritmetiska uttryck:

```
 2 + 3 * 4
(2 + 3) * 4
(a * 2) / 3
```

Observera att man måste skriva mellanslag mellan operander (variabler eller siffror eller funktionsanrop) och operatorer (* / + -). Alltså är 1 + 1 tillåtet men inte 1+1. Detta är för att det ska bli mindre klottrigt och mer lättläst.

Exemplen ovan är dock INTE tillåtna satser eftersom de beräknar ett värde, och sen struntar i det. Dock kan de användas i kombination med annat för att göra giltiga satser. Anledningen till att man gör en beräkning är ju för att man vill veta svaret och göra någonting vettigt med det, till exempel:

Tilldelning

Tilldelning (ge en variabel ett värde) görs med =. Det får inte förväxlas med matematikens = då denna fungerar annorlunda.

```
int a = 0
a = 5
```

a = 5 betyder att variabel a ska få värdet 5, eller "lägg siffran 5 i låda a". Man kan skriva det mesta på höger sida av = som kan räknas till ett värde, t.ex

```
a = 5 * (2 + 1)
```

Man kan även kombinera variabler och siffror, till och med variabeln vi ska ändra värde på:

```
int a = 2
a = a + 1
```

Detta skulle ju inte stämma i matte, men i programmeringssammanhang betyder det att a ska få värdet av a + 1. Eftersom a har värdet 2 blir a + 1 samma sak som 2 + 1 vilket blir 3. Då har vi kvar a = 3, alltså kommer värdet på a bli 3 när koden körs.

Predikatuttryck

Predikatutryck är kod som kan utvärderas till sant eller falskt. Till exempel är
2 > 1 sant och 1 > 2 falskt.

Det finns flera olika operatorer:

- == kontrollerar om det på vänstra sidan har samma värde som högra sidan (= betyder som vi nämnt tilldelning).
- > större än. a > b är alltså sant om a är större än b.
- < mindre än
- >= större eller lika mycket.
- <= mindre eller lika mycket.
- != sant om a INTE är lika mycket som b.

- && betyder "och", båda uttrycken på vänster och höger sida om && måste vara sanna för att hela villkoret ska bli sant. 3 > 2 && 5 != 5 blir inte sant eftersom bara 3 > 2 är sant.

- || betyder "eller", om ett eller båda uttrycken är sanna blir hela sant.

- 3 > 2 || 5 != 5 blir alltså sant.

If, else if, else

If, *else-if*, *else* satser består av logiska vilkor med kodblock som antingen körs eller inte körs beroende på om villkoren är sanna eller ej.

If, *else-if*, *else* satser är ganska simpla eftersom de är logiska och påminner om vanligt tal. *Om* det regnar ute ta paraply, *om* det snöar ute ta varma kläder *annars* ta vanliga kläder.

Den enklaste formen är att bara ha en *if* sats, som kan se ut så här:

```
int x = 5
int y = 0
if x > 2
  y = 3
```

Om x är större än 2 körs kodblocket. Eftersom det är fallet nu kommer alltså y ha värdet 3 efter att koden körts. Om x skulle vara mindre än (eller lika som) 2 händer ingenting. Vill man ändra på det kan man lägga till *else if* och/eller *else*.

```
int x = 1
int y = 0
if x > 2
    //x är större än 2
    y = y + x
else if x == 2
    //x är lika med 2
    y = x
else
  //x är mindre än 2
  y = y - x
```

Först kollar programmet om det första villkoret är sant. Om det INTE är det, kollas den första *else if* (den enda i det här fallet). Om ingen *if* eller *else if* är sann kommer alltid *else*-koden köras. En if-sats består alltså av

ett if-villkor, noll eller flera *else if*, och noll eller en *else*. Alla måste ha minst en rad kod som ska köras om villkoret uppfylls.

While-loop

En *while*-loop består av ett villkor (predikatutryck) och ett block med kod. Kodblocket körs om och om igen tills villkoret blir falskt.

```
int x = 9
while x > 2
        //kommer köra koden nedan tills x > 2 blir falskt
        x = x - 2
```

`x > 2` är alltså vilkoret som måste bli "False" för att loopen ska avslutas.

`x = x - 2` är alltså kodblocket som kommer köras om och om igen tills tills vilkoret blir falskt.

For-loop

En *for*-loop är ett block av kod som körs ett visst antal specifierade gånger. For loopen har samma syfte som while-loopen.

En *for*-loop består av en (int) variabel, startvärde, stoppvärde och ett kodblock. Detta möjligör att ett kodblock körs ett bestämt antal upprepade gånger.

När man skriver en *for*-loop anger man ett startvärde, ett stoppvärde och ett kodblock som ska köras.

```
int a = 5
for 0 to 3
   //denna kod kommer köras 3 gånger
     a = a + 2
```

`for 0 to 3` är antalet gånger du vill att kodblocket ska köras, i detta fall 3 gånger. `a` får alltså värdet 11.

`a = a + 2` är kodblocket som kommer köras 3 gånger i programmet.

0

a får värdet 7 (5 + 2)
1 (0 + 1)
a får värdet 9 (7 + 2)
2 (1 + 1)
a får värdet 11 (9 + 2)
3 (2 + 1)
Här händer inget mer eftersom loopen är klar.

Ibland kan man vilja hålla reda på vilket steg i loopen man är. Det brukar göras med en iterator (som oftast kallas i, men man kan ha vilket variabelnamn du vill). Iteratorn "deklareras" direkt i loopen så den behöver man inte deklarera innan. Man kan till exempel skriva:

```
int a = 5
for i from 0 to 3
    a = i + 3
```

i kommer alltså få värdena 0, 1, 2 och 3 vilket gör att a kommer bli 3, 4 och 5 (a blir aldrig 6 eftersom loopen inte körs när i = 3).

Christer Thor
chrth485@student.liu.se

```
int a = 0
for i from 0 to 5 jump 2
  a = a + 1
```

`jump 2` gör att den kommer att öka värdet på `i` med 2 varje gång loopen körs.

i = 0
a får värdet 1 (0 + 1)
i = 2 (0 + 2)
a får värdet 2 (1 + 1)
i = 4 (2 + 2)
a får värdet 3 (2 + 1)
i = 6 (4 +2)

Detta gör att den bara kommer köras 3 gånger eftersom i är större än 5 redan efter 3 gånger.

I *for*- och *while*-loopar vill man ibland kunna avbryta loopen eller hoppa över något steg. Skriver man *break* i en loop kommer man hoppa ur loopen. Skriver man *next* i en loop hoppar den direkt till nästa iteration.

```
int a = 0
//true blir ju alltid sant, vilket gör att loopen aldrig
slutar om man inte hoppar ur den med break
while true
    a = a + 1
    if a == 2
        next
    if a > 1
        break
```

När a blir 2 kommer den hoppa direkt härifrån till hit. Först nästa gång när a blir 3 kommer programmet se att `a > 1` och avbryta loopen.

Funktioner

En funtion är helt enkelt ett eget program som du kan kalla på i ditt program för att slippa skriva om saker överallt i koden. Detta gör programmet mer lättläst och uppdelat. Det blir även lätt att skriva kod tillsammans med andra, då ni kan skriva egna funktioner och sedan sätta ihop dem.

En funktion innehåller alltid ett kodblock som kan vara godtyckligt stort och utformas så den kan ta in olika slags parametrar eller inga parametrar.

För att skapa en funktion behöver du först definera den och välja ett passade namn sedan även välja om den ska ha parametrar eller ej.

Det finns även inbyggda funktioner i pLite. En heter `help()` och listar alla inbyggda funktioner. print() är en funktion som tar in en text som argument (förklaras längre ner) och skriver ut texten på skärmen.

```
def test_funktion()
  print("Här inne i kodblocket kan vi göra vad vi vill")
    print(vi kan även kalla på andra funktioner etcetera")
```

`def` här meddelar vi att vi vill definiera en ny funktion så programmet vet att den finns och kan kalla på den senare.

`test_funktion` här skriver vi ett passande namn på funktionen som förklara vad den har för syfte och så att vi kan kalla den sedan genom detta namn.
Dessa är exempel på kodblock man kan använda sig av. All kod som du kan skriva utanför funktioner kan du skriva inuti dem.

För att sedan kalla på funktionen skriver man bara `test_funktion()`. Uppmärksamma `()` i slutet av namnet vilket behövs för att exekvera funktionen.

Man kan även skriva funktioner med en eller flera inparametrar (tänk variabelnamn).

```
def test_funktion(tal, text)
```

Nu har vi även angett att funktionen tar emot två argument (tänk värden på variabler). När man anropar funktionen skriver man in värdena man vill ska tilldelas funktionens två variabler, `tal` och `text`.

Om funktionen ser ut så här:

```
def test_funktion(tal, text)
    for 0 to tal
        print(text)
```

kommer den att skriva texten så många gånger som `tal` är, om man till

exempel skriver:

```
test_funktion(3, "Hello World!")
```

kommer utskriften av detta bli nedanstående:

```
Hello World!
Hello World!
Hello World!
```

Returvärden

En funktion kan även returnera ett värde. Ungefär som att du kan få ut ett värde från en variabel, kan du även få ut ett värde från en funktion.

```
def f(x)
    return x * 2 - 1

print( f(2) )
```

`f(2)` får alltså värdet 2 * 2 - 1 = 3, och print-funktionen skriver alltså ut 3 på skärmen. Funktionsanrop kan användas på alla ställen där man kan använda värdet på en variabel.

Indentering

Bra indentering!

```
def good_test(x)
    print("Hej detta är bra indenterat")

    if x > 4
         print("x är större än 4")

good_test(5)
```

Dålig indentering!

```
def bad_test()
   print("Hej detta är dåligt indenterat")
if x > 4
print "x är större än 4"
```

Indentering innebär att man gör ett indrag i början av raden. I

programmering brukar man indentera koden för att den ska bli mer lättläst och översiktlig. I pLite är det även indenteringen som avgör vilken kod som tillhör vad.

Allting som innehåller en eller flera rader kod (funktioner, if-satser, loopar) ska ha den innehållande koden indenterad. Detta avgör vad som t.ex. tillhör en funktion och vad som ligger utanför funktionen.

Om man som ovan har t.ex. en *if*-sats i en funktion så ska all kod som tillhör *if*-satsen vara indenterad ett steg extra.

I det dåliga exemplet ovan skulle funktionen bara innehålla print-raden, if-satsen skulle ligga utanför och eftersom den sista kodraden inte är indenterad skulle if-satsen inte ha nån kod att köra om den var sann, vilket inte är tillåtet att skriva.

Den vanligaste indenteringen är 4 mellanslag per "indrag", fast det går att ha ett annat antal, bara du är konsekvent.

Räckvidd

Räckvidd, eller Scope som det heter på engelska, innebär att variabler "når" olika långt. I pLite innebär det att alla variabler som skapats på en viss indenteringsnivå tas bort när du går tillbaks till en tidigare nivå. Rent praktiskt blir det så att om du deklarerar en variabel i en funktion finns den i funktionen men inte utanför. Gör du en ny variabel i en *for*-loop i funktionen finns den i loopen men inte utanför. Du får deklarera fler variabler med samma namn om de är på *olika* nivåer. Försöker du använda det variabelnamnet får du den variabel som deklarerades på den djupaste nivån. Variabler som är deklarerade på den första nivån brukar kallas globala variabler, medan andra brukar kallas för lokala variabler. Allt detta kan låta rörigt och krångligt, men är ganska enkelt och logiskt när man vant sig.

```
int x = 0
def scope_test()
    int x = 1

    if x == 1
        x = 2

    print(x)

print(x)
scope_test()
print(x)
```

Utskriften blir

```
0
2
0
```

Den första print-funktionen skriver ut det globala x:et, eftersom det andra inte existerar utanför funktionen. Sedan anropar vi funktionen scope_test()som sätter funktionens x till 2 och skriver ut det eftersom det är djupare nästlat än det globala x:et. Den sista print-funktionen skriver ut 0 eftersom vi aldrig ändrade värdet på det globala x:et!

Systemdokumentation

BNF-syntax

```
PROGRAM          : STMT [T STMT]*

T                : "\n"

STMT             : T
                 | "none" T
                 | "break" T
                 | "next" T
                 | RETURN_EXPR T
                 | IF_EXPR
                 | FOR_EXPR
                 | WHILE_EXPR
                 | FUNCTION
                 | VAR_DECL T
                 | ASSIGN_EXPR T
                 | FUNC_CALL T

IF_EXPR          : "if " PREDICATE_EXPR T INDENT
                       STMT+ DEDENT
                   ["else if " PREDICATE_EXPR T INDENT
                       STMT+ DEDENT]*
                   ["else " T INDENT
                       STMT+ DEDENT]?

FOR_EXPR         : "for" [VAR "from"]? VALUE "to" VALUE
                   ["jump" VALUE]? T INDENT
                       STMT+ DEDENT

WHILE_EXPR       : "while" PREDICATE_EXPR T INDENT
```

```
                    STMT+ DEDENT

FUNCTION        : "def" VAR "(" [VARS]? ")" T INDENT
                    STMT+ DEDENT

RETURN_EXPR     : "return" :VALUE

VAR_DECL        : "bool"   VAR " = " PREDICATE_EXPR
                | "int"    VAR " = " ARITM_EXPR
                | "float"  VAR " = " ARITM_EXPR
                | "string" VAR " = " STRING

ASSIGN_EXPR     : VAR "=" ARTIM_EXPR

VALUE           : VAR
                | FUNC_CALL
                | INTEGER
                | FLOAT

FUNC_CALL       : VAR "(" [ARGS]? ")"

PREDICATE_EXPR  : BOOL
                | VAR
                | ARITM_EXPR COMPARE_OP ARITM_EXPR
                | PREDICATE_EXPR LOGIC_OP PREDICATE_EXPR

COMPARE_OP      : "<"
                | "<="
                | ">"
                | ">="
                | "!="

LOGIC_OP        : "&&"
                | "||"

ARITM_EXPR      : VALUE [OPERATOR VALUE]*
                | "(" ARITM_EXPR ")"

OPERATOR        : "+"
                | "-"
                | "*"
                | "/"

VARS            : VAR ["," VAR]*

VAR             : LETTER (LETTER | DIGIT | "_")*

ARGS            : ARG ["," ARG]*
```

```
ARG             : STRING
                | BOOL
                | ARITM_EXPR

LETTER          : [a-zA-Z]+

DIGIT           : ["0" | NON_ZERO_DIGIT]+

NON_ZERO_DIGIT  : [1-9]

BOOL            : "True"
                | "true"
                | "False"
                | "false"

INTEGER         : ("-"? NON_ZERO_DIGIT DIGIT*)
                | "0"

FLOAT           : INTEGER "." DIGIT+

STRING          : " {any_char} "
                | ' {any_char} '

INDENT          : "__indent__"

DEDENT          : "__dedent__"
```

Översikt över pLite

Språket är en lättare variant av "traditionella" programmeringsspråk och består av 3 delar i 3 filer, en lexer, en parser och en nodbyggare/ syntaxträdsbyggare.

Lexer

Lexern tar in en textsträng med hela koden som ska parsas. Med hjälp av massa reguljära uttryck plockar den ut tokens och lägger in dom i en array (@token_list). Den matchande texten plockas bort ur kodsträngen och sen börjar den om och fortsätter så tills strängen är tom.

När en newline läggs till i @token_list sätts variabeln found_newline till true. Nästa iteration kommer lexern att räkna antalet mellanslag i början av strängen. Är det fler mellanslag än sist läggs en indenteringstoken in i token-arrayen, och den nya indenteringsnivån sparas i @indent_stack. Är den nya nivån mindre går programmet igenom @indent_stack och

genererar rätt antal dedent-tokens beroende på vilken nivå som matchades.

Parser

Vår parser bygger på rdparse. När regler matchas skapar vi instanser av nod-klasser som är definierade i node_maker.rb. När allting matchats klart körs vår evaluate-funktion för vår instans av Stmt_list_node, som anropar evaluate på alla statements, som i sin tur anropar vidare ner i syntaxträdet.

Matematikreglerna tog vi i huvudsak från klassen DiceRoller i rdparse.rb (en fil vi fick använda oss av i kursen). Reglerna är skrivna på ett sånt sätt att + och - matchas före * och /, och uttryck inom parenteser matchas först. Vid t.ex. uttrycket 1 + 2 * 3 skapas en Aritm_node med en Constant_node med värde 2, operatorn + och en ny Aritm_node som i sin tur innehåller 2, * och 3.

Syntaxträd

Varje nod-klass har minst 2 funktioner: en initialize (som körs när klassen skapas) där den tar in de värden som behövs och en evaluate som utvärderar koden (och ofta anropar evaluate för andra instanser).

Erfarenheter och reflektion

Från början kändes det som att det skulle bli oerhört svårt, skapa ett eget språk från grunden verkade vara långt ifrån vår nivå och lite skrämmande, speciellt då man fortfarande bara hade erfarenheter av BNF-syntaxer som vi snabbt gick igenom i vår första kurs på universitet då vi lärde oss Python.

När vi skriv vår språkspecifikation kom vi tidigt överens om vad vi ville göra för språk och vad vi hade för vision om det. Detta gjorde att väldigt många designproblem som uppstod mitt i arbetet av projektet kunde lösas snabbt och enkelt.

Print-funktionen är ett bra exempel nedanför.

```
print "utan parenteser"
```

eller

```
print("med parenteser")
```

Detta problem löstes väldigt snabbt och smidigt genom att tänka efter hur vår vision var, och då dra slutsatsen att vi blir tvunga att använda paranteser. Vi vill att man ska lära sig att print är en funktion som tar det man skriver inom parenteser som argument.

Det blev också mycket enklare då vi fick reda på att allt inte skulle skrivas från scratch utan vi fick ta hjälp av rdparsern om man valde detta. Vi valde att utgå från rdparsern och sedan skriva vår egen lexer. Detta gick bra trots att det uppstod några problem. Problemen gav oss inga större svårigheter och i helhet flöt det på riktigt bra.

Något som gav oss mer krångel än väntat var när lexern var klar och vi fick in alla tokens vi ville.

Hur skulle hela uppbyggnaden och evalueringen ske? Vad var "noder" för något, hur skulle de fungera och framförallt hur skulle vi koda dom? Vi tog en sak i taget och med hjälp och förklaringar från Haraldsson och klasskompisar löste det sig bättre än vi vågat hoppats på.

Angående ändringar i språkspecifikationen har det inte blivit några större skillnader. Vi har ett språk som är tänkt för nybörjare och ville hålla nivån väldigt simpel samt se till att vi hade med grunderna i programmeringen. Syftet med det är att nybörjare ska lära sig om programmering utan att bli förvirrade eller att det blir alldeles för mycket på en gång.

Den enda stora ändringen som inte kommer bli fullständig så som vi vill ha den är våra errormedelanden då tiden tyvärr tog slut. Vi ville från början skapa en väldigt simpel men effektiv errormeddelare för våra användare så de lätt kan förstå vart och varför det blev fel. Detta är något vi tycker många språk inte lyckats med.

De erfarenheter vi framförallt har fått är att vi lärt oss väldigt mycket om just Ruby allt från dess invecklade egenskaper till lättare och på hur många olika sätt man kan lösa olika problem i Ruby genom att varje dag ständigt utforska språket. Vi har också fått erfara Rubys svårare sidor och nackdelar med språket.

Vi har även lärt oss att jobba under tidpress då vi har haft deadlines att följa hela tiden. Vi fick även många goda råd genom att ha kontakt med en lärare som hade översikt över projektet. Genom den muntliga kommunikationsdelen har vi även fått lära oss vad man skall tänka på när vi redovisar sådana här projekt och överhuvud taget, samt vad vi ska undvika att göra för att få bästa möjliga resultat.

Vi har dessutom fått en rejäl inblick i hur ett programmeringsspråk är

uppbyggt och vilka svårigheter som finns. Vi fick flera "aha"-upplevelser under projektets gång då många saker vi tagit för givna när vi programmerat i andra språk visade sig vara allt annat än triviala. Framförallt många syntaxregler vi tyckt varit konstiga förut känns mer logiska nu då vi förstår varför de behövs. Det var spännande och lärorikt att se "bakom kulisserna" på ett programmeringsspråk.

Referenser

- http://ruby-doc.org/
    - Datum: 26/3 - 12/5

- http://www.ida.liu.se/~TDP019
    - Anders Haraldssons seminarier och möten i kursen TDP019
    - Datum: 19/3, 26/3, 14/4, 21/4, 28/4, 5/5, 12/5