Web Presentation Generator

WPG

Ett projekt skrivet av Mathias Pettersson och Jim Torarp

Förord

Vi vill tacka alla lärare som har givit oss kunskapen för att kunna genomföra detta projekt. Vi är nöjda med våran presentation i denna kurs och resultatet vi har uppnått och kan med glädje presentera vårat egna programmeringsspråk.

Inledning

Vi har gått en projektkurs ”datorspråk” (TDP019) som är en del av IP-programmet på Linköpings universitet (LiU). Kursen har varit uppdelat i olika examinationsmoment, bl.a. en programmeringsdel där vi skulle implementera ett eget programmeringsspråk, samt även några moment där vi skulle framföra olika presentationer.

WPG, eller *Web Presentation Generator* är vårat programmeringsspråk som vi tog fram för att på ett enkelt sätt kunna presentera text i HTML format.

Vi strävade efter att ta fram ett språk som inte skulle vara allt för svårt för att använda sig av där ingen förkunskap i HTML skulle vara ett krav.

Innehållsförteckning

Examinator:
Anders Haraldsson

TDP019: Projekt: Datorspråk
Versionsdatum: 2010-05-24

Mathias Pettersson, matpe860
Jim Torarp, jimto948

Användarhandledning

Kort om språket

Språket Web Presentation Generator (*WPG*) är skrivet i programmeringsspråket Ruby. Språkets syfte är att vara ett lättandvänligt språk för att skapa/generera enkla htmlsidor i form av presentationssidor.

Målgruppen för språket är i blandad ålder och det behövs inga speciella förkunskaper i html för att lära sig vårt språk.

1.1 Hur språket används

Koden skrivs i en vanlig textredigerare (t.ex. notepad++) som sedan sparas ner i filformatet "*.wpg*". Koden från filen exekveras via programmeringsspråkets kompilator och gör då om koden till godtycklig htmlkod. För att exekvera en *.wpg*-fil görs detta enklast genom att installera programmet **SciTE** för att få alla nödvändiga plugins som behövs.
När SciTE finns installerat så måste filen "*wpg_language.rb*" öppnas i programmet för att sedan editera längst ned i koden så att rätt fil exeveras. Detta görs längst ner i den filen där det står "*EXECUTE YOUR FILE*", sen så är det bara att följa instruktionerna som står där.

Med språket kan går det att bl.a. skriva ut text, utföra enklare beräkningar med multiplikation, division, addition och subtraktion där även prioritering inom parentes fungerar bra, m.m.
Indentering av kod spelar ingen roll, istället avslutas ett kodblock eller en sats med *end*.

Notera även att det är tillåtet att allt skrivs på en enda lång rad i språket, dock blir det mera lättläst om satserna separeras med ny rad och indentering följs enligt exemplen.

2.1 Skapa kodblock

För att öppna ett kodblock används följande rad rad: html("*filnamn*") begin.
Där *filnamn* kan bytas ut mot godtyckligt antal tecken (det rekommenderas dock att tecken som a-z, A-Z, 0-9, -, _, . används) som sedan blir resultatet i form av en *html*fil efter exekvering.
Nu är kodblocket öppet och kod kan börja skrivas. När koden är skriven måste kodblocket stängas m.h.a. *end.*

Exempelkod:

```
html("MyFirstHtmlFile") begin
        din kod skrivs här.....(se 2.1)
end
```

Nu kan koden börja skrivas, fortsätt till avsnittet *2.1 kodning* för att läsa mer om hur koden skall skrivas för WPG.

2.1 Kodning

En kodrad avslutas alltid med ";" om det inte är en *loop* eller en *kontrollsats*. *End* avslutas inte heller med ";".

2.2 Kommentarer

En kommentar är ett stycke text som inte tolkas av programmet. Detta kan användas för att göra minnesnoteringar eller ta bort kod som inte skall användas för tillfället.
Om en kommentar bara skall vara över en rad används följande sätt.
Exempelkod:

```
# en enradskommentar
```

Men kommentarer kan också skrivas över flera rader och då används /* för att indikera var kommentaren börjar och */ för att visa vart kommentaren slutar.
Exempelkod:

```
/*
variabel = "test string";
summa = variabel+variabel;
print(summa);
*/
```

2.3 Typvärden

Strängar är en text med godtyckligt antal tecken innanför citattecken. Detta skrivs på följande sätt:

Exempelkod:

```
"Detta är en sträng";
```

Heltal skrivs som vanligt:

Exempelkod:

```
5;
```

Decimaltal separeras med en *punkt* mellan heltalet och decimaltalet:

Exempelkod:

```
5.5;
```

Sanningsvärden sant och falskt, skrivs med gemener:

Exempelkod:

```
# sant
true;
# falskt
false;
```

Listor kan också användas och de är till för att samla mer data på ett ställe.
En lista är en sekvens med värden, den kan innehålla bl.a. variabler, heltal, en till lista (lista i lista) mm.
För att visa att det är en lista så skrivs värdena mellan hakparenteser [och] där värdena (elementen) separeras med kommatecken. Om listan bara innehåller ett värde så används inga kommatecken.
Exempelkod:

```
# Här är en lista som innehåller: sträng, heltal, lista (en lista i en lista)
["En sträng", 1234, ["En array", "ett till element"]];
```

Det går att *indexera* en lista för att hämta ut enstaka element ur den. Detta görs m.h.a. hakparenteser och ett heltal.
Exempelkod 2:

```
# Här är en lista som innehåller: sträng, heltal, lista (en lista i en lista)
lista = ["En sträng", 1234, ["En array", "ett till element"]];
# Här hämtas första elementet
lista[0];
```

Notera att en lista börjar på *index 0* och slutar på antalet element i listan-1, d.v.s. 2 i detta fall.

2.4 Satser

Tilldelning
En tilldelning sker till en s.k. *variabel.* En variabel blir en slags nyckel som kan användas för att spara ner information och sedan hämta ut den igen.
Ett variabelnamn måste börja med en bokstav a-z (gemener) följt av godtyckligt antal bokstäver a-z eller underlinje (eng. *underscore*) "_".
Tilldelning sker genom att definiera variabeln först enligt ovan sedan kommer ett likhetstecken "=" för att visa vilket värde variabeln skall få, följt av *typvärdet.*
Det går bra att spara ner alla typvärden till en variabel, samt funktionsanrop eftersom dom har sk. *returvärden*.

Exempelkod:

```
variabel1 = "Sträng";
variabel2 = 123456;
variabel3 = ["en array", "ett element"];
variabel4 = (2*4);
variabel5 = read("testfil.txt");
variabel6 = min_funktion("en inparameter");
```

Loopar (slingor)

En loop är en sats som utför ett visst antal instruktioner (satser) ett X antal gånger som användaren definierar.

Loopar börjar med nyckelordet *for* följt av en parentes. Innanför parentesen så definierars först en variabel (enligt ovan) som loopen kommer att "loopa" på. Därefter kommer nyckelordet *in*, samt hur många gånger loopen skall köras. Detta kan anges på två olika sätt.

Aningen anges ett positivt heltal från 0 och större följt av två punkter och avslutas med ett till heltal större eller lika med än 0, exempelvis:

1..4

Detta gör att loopen kommer att köras från 1 till 4, d.v.s. 4 gånger totalt. Därefter stängs parentesen och nyckelordet *do* skrivs ut.

Nu är loopen öppen och det går att skriva ett godtyckligt antal satser. Loopen avslutas med nyckelordet *end* för att indikera att loopen är slut.

Eller så anges en lista sträng och då kommer loopvariabeln innehålla dom olika elementen i listan från början till slut. Även en sträng kan anges och då kommer loopvariabeln innehålla varje bokstav i strängen, från början till slut.

Exempelkod 1 (lista):

```
# Denna kod skriver ut varje element i variabeln array
array = ["ett element", "ett till element", "sista elementet"];
for(element in array) do
        print(element);
end
```

Exempelkod 2 (given sekvens):

```
# Denna kod skriver ut "Hej" 5 gånger
for(1..5) do
        print("Hej");
end
```

If-satser (logisk kontrollsats)

En kontrollsats används för att undersöka om ett logiskt uttryck är sant (*true*).

Om dom är det så kommer satsen att köras, om inte så kommer den kontrollsatsen att "hoppas över".

En kontrollsats börjar alltid med "if" följt av parenteser där du skriver ett logiskt uttryck. Dessa logiska uttryck går att använda:

"=="
"<"
">"
"<="
">="
"!="

"and"
"or"

Du kan använda en "else" i en ifsats men det är inte ett måste, du kan också använda hur många "elseif" som helst.

Exempelkod:
```
If(a == 5) then
        print("A har värdet 5")
end
```

Exempelkod2:
```
if(a== 10) then
        print("A har värdet 10");
else
        print("A har annat värde:");
        print(a);
end
```

Exempelkod3:
```
if(a== 10) then
        print("A har värdet 10");
elseif(a==5) then
        print("A har värdet 5");
else
        print("A har annat värde:");
        print(a);
end
```

2.5 Inbyggda funktioner:

En inbyggd funktion är en funktion som alltid ser likadan ut, men kan ge olika returvärden beroende på vad in parametrarna är.
Utryck är en sammansättning av typvärden/satser du kan använda +, -, *, / samt parenteser både på strängar, siffror. Mycket htmlkod kan nu med hjälp av loopar(slingor) samt if-satser(kontrollsatser) skrivas på ett mycket effektivare och kompakt sätt.
set_img() används för att skriva ut en bild till resultatfilen. Inom parenteserna måste en *sträng* eller en *variabel* som refererar till en bilds sökväg anges.
Exempelvis så betyder sökvägen "bilder/mathias.jpg" att bilden *mathias.jpg* ligger i mappen *bilder* (om vi utgår från katalogen som språkfilen ligger i), medan sökvägen "../mathias.jpg" betyder att bilden *mathias.jpg* ligger i katalogen ovanför.
read() läser in en fils samtliga rader och du kan spara ner all information i en variabel. Read används genom att skriva "read" följt av två parenteser. I parentesen skriver man den fil som skall

öppnas inom citattecken som exemplet nedan. Read används för att läsa in data från filer med text i.

Exempelkod:
```
Variable = read("testfil.txt")
```

open() läser in text från en sk. "databasfil" som är uppbyggd på ett speciellt sätt. Du börjar med att skriva "open" följt av två parenteser. Inom parenteserna skrivs namnet på den databas som du vill läsa ifrån.

Börja med att tilldela en variabel attributet "open":
```
db = open("mydata.db");
```

Där "mydata.db" kan bytas ut mot filnamnet på den aktuella databasfilen.
Ett exempel på en databasfil är:
```
[person1]
name Jim
age 20
[/person1]
[person2]
name Mathias
lastname Pettersson
age 20
[/person2]
```

Där [person1] blir en tabell som innehåller "nycklarna": *name* och *age.* Medan [person2] har nycklarna, *name*, *lastname* och *age.* Tabellerna avslutas med "/" framför öppnings-blocket, ex: [/person1]. Alltså, alla nycklar måste vara deklarerade innanför [person1] *här* [/person1] för att dom skall registreras.

Tabellerna behöver *inte* ha samma nycklar, som kan ses i exemplet ovan.
För att hämta ut en tabell så kan detta exempel följas (om vi fortsätter på tidigare exempel):

Exempelkod 2:
```
person_one = db["person1"];
```

Nu innehåller variabeln *person_one* alla nycklar till person1 i databasen i forma av en lista.
För att skriva ut exempelvis värdet för nyckeln *name* så kan följande exempel följas:

Exempelkod 3:
```
name = person_one["name"];
print(name);
```

Nu innehåller variabeln *name* värdet för tabellen *person1* och nyckeln *name* i databasfilen.
Det går även bra att hämta en nyckel från en tabell direkt efter att databasfilen har öppnats.

Exempelkod 4:
```
name = ["person1"]["name"];
```

print() används för att skriva ut saker på din htmlsida. Du kan skriva ut strängar, siffror och variabler. Du skriver först det du vill skall skrivas ut sedan kan du skriva "," sen ":" och någon färg för att färglägga texten du skriver ut i hmtl. Default färgen är svart om ingen färg anges.
Färger:
:BLACK
:GREEN
:BLUE
:PURPLE
:YELLOW

Exempelkod:
```
print("Detta är en sträng");
print(5);
Variable = "Hej";
print(Variable);
```

len() undersöker längden på det som skrivs mellan parenteserna, Exempelvis längden på en sträng "jim" resulterar i en heltals siffra 3, medan längden på ett heltal 1337 resulterar i heltalet 4.

Exempelkod:
```
# Detta kommer att resultera i heltalet 34 (mellanrum räknas in)
len("Ta reda på längden av denna sträng");
```

2.6 Definiera egna funktioner:

En funktion är uppbyggd på detta sätt: **def funktionsnamn(***eventuella inparametrar***) begin** *deklarieringar* **returvärde end**.
Funktioner inleds alltid med *def* före funktionens namn för att definiera att det är just en funktion.
Efter *def* skrivs funktionens namn som skall vara i versaler.
Sedan kommer två parenteser där det går att deklarera *in parametrar* till funktionen (d.v.s. *variabler* som skickas in till funktionen vid *funktionsanrop*) som separeras med kommatecken. Att ange in parametrar är dock inget som är obligatoriskt.
Efter parenteserna kommer ett till nyckelord som måste skrivas och det är *begin*.
Nu är funktionen öppen och användaren kan skriva godtyckligt antal satser.
När alla satser är angivna så måste ett *returvärde* anges med nyckelordet *return* innan returvärdet.
Ett returvärde kan bestå av bl.a. *variabler, tal, strängar o.s.v.*

Efter att returvärdet har angivits så måste funktionsblocket stängas, detta görs med det vanliga nyckelordet *end*.

Exempelkod 1:
```
def funktions_namn(inparamet_one, inparameter_two) begin
        print(inparameter_one, :GREEN);
        return (inparameter_one + " " + inparameter_two);
end
```
En eller flera inparametrar är inte nödvändigt, språket tillåter att deklarera en funktion utan inparameter.

Exempelkod2:
```
def funktions_namn() begin
        print("Mathias Pettersson");
        return 0;
end
```

2.7 Funktionsanrop

Ett funktionsanrop görs när en funktion skall exekveras (utföras).
För att anropa en funktion måste funktionen vara definierad.
Ett funktionsanrop kan se ut så här:

Exempelkod 1:
```
FUN("Mathias", "Pettersson");
```
Här anropas funktionen *FUN* med parametrarna *"Mathias*" och *"Pettersson*".
Det går även att tilldela en variabel variabel en funktion, då kommer variabeln att få funktionens returvärde.
Om vi utgår från funktionen i *exempel 1* punkt *2.1.5* så kommer följande exempel att skriva ut texten "*Mathias Pettersson*" (förutom att funktionen i sig skriver ut texten "*Mathias*") i htmlfilen.

Exempelkod 2:
```
result = FUN("Mathias", "Pettersson");
print(result);
```

3.1 Databaser

En databas är en fil där information kan lagras separat från koden. Detta gör det smidigt då det minskar mängden information i koden så det t.ex. blir mer lättläst. En databas består av *tabeller* där varje tabell har sina egna unika *nycklar* med information.
Att skriva en egen databas görs enklast genom att öppna en texteditor, exempelvis "*anteckningar*" för Windows eller "*Gedit*" för Linux och sedan skapa ett tomt dokument.

För att skapa en tabell i databasen måste den öppnas med ett nyckelord som sedan blir namnet för tabellen, nyckelordet får innehålla godtyckligt antal tecken, helst a-z, A-Z, 0-9, - eller _. Namnen bör vara unika för tabellerna för att undvika kollision med andra tabeller i databasen.
När en ny tabell öppnas så skrivs tabellnamnet inom hakklamrar, t.ex.

```
[person1]
```

Nu är tabellen öppen och det går bra att lägga till nycklarna med tillhörande information.
För att lägga till en nyckel med ett värde så skrivs först nyckeln (som senare används för att referera till värdet) sedan värdet för nyckeln, t.ex.

```
name "Mathias"
```

Nu har nyckeln *name* värdet *Mathias*.
Notera att inget likhetstecken skrivs ut eller att inget semikolon används, samt att strängar fortfarande skrivs med citattecken. Heltal, decimaltal o.s.v. skrivs precis som i programmeringsspråket (d.v.s. utan citattecken för att fungera som tal och inte strängar).
När alla nycklar är tillagda för tabellen så måste den stängas. Detta görs genom att göra likadant som vid öppningen, fast med ett snedstreck (slash/frontslash) före namnet, t.ex.

```
[/person1]
```

Notera att samma namn som databasen öppnades med, måste även användas för att stänga den.
En komplett databas kan likna:

Exempelkod 1:
```
[person1]
firstname "Mathias"
lastname "Pettersson"
fullname "#{firstname} #{lastname}"
image "mathias.jpg"
[/person1]
[person2]
firstname "Jim"
lastname "Torarp"
fullname "#{firstname} #{lastname}" (Här kommer nyckeln fullname att ha värdet av firstname och lastname, d.v.s. "Jim Torarp")
[/person2]
```
Viktigt: det finns inga kommentarer för databaser, dock om text skrivs utanför tabellerna så kommer den inte att registreras.

Systemdokumentation

Introduktion

En snabb överblick

Detta dokument är till för Dig som är intresserad av att ta reda på mer information om WPG och hur språket är uppbyggt samt även vilka beståndsdelar systemet består av och hur det fungerar.

Vad är WPG?

Förkortningen WPG står för *Web Presentation Generator*, som är ett språk för att skapa och generera enklare presentationer i HTML form. Ingen förkunskap om hur HTML fungerar eller hur det skrivs är nödvändigt.

Teknisk specifikation

Klasser och dess relationer

File_node

initialize(file, nodes)	Tar ett filnamn och en array med noder.
write(value)	Tar emot ett värde som sedan skrivs till en angiven fil.
match_link(str)	Matchar en sträng efter http:// adresser och gör om dessa till klickbara länkar.
match_email(str)	Matchar en sträng efter mejladresser och gör om dessa till klickbara länkar.
array_to_string	Gör om en array till en sträng som liknar en array vid utskrift.
hash_to_string	Gör om en hash till en sträng som liknar en hash vid utskrift.
open	Öppnar filen som skall skrivas till
close	Stänger filen som skrivs till.
eval	Evaluerar all genererad programkod (alla noder).

Evaluerar hela programmet. Tar emot en Array som innehåller alla noder som skapats i korrekt ordning.

Constant_node

initialize(value)	Tar ett konstant-värde, t.ex. en *String* eller *Integer.*
eval	Returnerar konstanten som gavs vid initieringen.

Klassen används för att skapa konstanta värden som t.ex. strängar, heltal, flyttal o.s.v.

Array_index_node

initialize(value, index)	Hämtar ut ett index ur en array.
eval	Returnerar värdet på valt index.

Används för att kunna hämta ut specifika värden ur en array i vårat programspråk.

Not_node

initialize(value)	Tar emot ett sanningsvärde, eller logiskt uttryck.
eval	Returnerar negationen på det angivna logiska värdet.

Gör negation på ett logiskt uttryck eller ett sanningsvärde, exempelvis *true* blir *false* och *true and false* blir *true*.

Variable_node

initialize(id)	Tar ett namn som inparameter och skapar en variabel.
eval	Returnerar variabeln.

Initieringen skapar en variabel och evalueringen hämtar ut värdet på variabeln från en specifik hash som innehåller programmets alla variabler.

Print_node

initialize(value, color)	Tar emot ett värde samt en färg för utskrift.
eval	Returnerar en array med värdet som skall skrivas ut och färgen det skall skrivas i.

Klassen används för att kunna skriva ut olika delar i programspråket. En kontroll i *File_node:s* evalueringsmetod görs för att kontrollera om ett objekt skall skrivas ut.

Loop_array_node

initialize(type, variable, arr, data)	**type** är vilken sorts loop (*for* i detta fallet). **variable** är vilken loopvariabel som loopen kommer använda sig av. **arr** är arrayen som loopen kommer att gå igenom. **data** är alla satser som loopen innehåller.
eval	Stegar igenom arrayen som angivits elementvis och ändrar loopvariabeln till aktuellt värde från arrayen.

Används för att få ut varje enskilt värde ur en array.

Loop_node

initialize(type, variable, rounds, data)	**type** är vilken sorts loop (*for* i detta fallet). **variable** är vilken loopvariabel som loopen kommer använda sig av. **rounds** bestämmer hur många gåger en loop skall exekveras. **data** är alla satser som loopen innehåller.
eval	Satserna utförs x antal gånger utefter angivet värde i **rounds**.

Används för att utföra ett visst antal satser ett x antal gånger.

Logic_stmt_node

initialize(level, expr, data)	Tar emot viken sorts kontrollsats det är, ett logiskt uttryck och alla satser som angivits. innuti kontrollsatsen.
eval	Utför alla satser om det logiska uttrycket som angivits är sant.

Klassen används för att skapa logiska kontrollsatser så som *if, elseif* som alltid tar en logisk kontrollsats och *else* som alltid kommer att evaluera sant.

Logic_stmt_block

initialize(stmts)	**Stmts** är en lista som innehåller godtyckligt antal *Locic_stmt_node:s* (logiska kontrollsatser).
eval	Går igenom listan med logiska kontrollsatser och evaluerar det första som har ett logiskt uttryck som är sant.

Samlar på sig alla logiska kontrollsatser som måste ha en if-sats i toppen, följt av godtycklig antal elseif-satser, och/eller en else-sats.

Assign_node

initialize(var, expression)	Tar emot en *Variabel_node* och ett värde, t.ex. en *Expr_arithm_node* eller en Constant_node.
eval	Lägger till variabeln med tillhörande värde i en specifik hash som innehåller programmets alla variabler.

Klassen används för att tilldela en variabel ett värde i vårat programspråk.

Logic_arithm_node

initialize(logic, op1, op2)	Tar emot en logisk operator, samt två operander (logiska sanningsvärden i form

av *Constant_node*).

eval	Returnerar resultatet av det logiska uttrycket.

Skapar ett logiskt uttryck, med exempelvis operatorerna: *or, and, <, >, != o.s.v.*

Expr_arithm_node

initialize(op, op1, op2)	Tar emot en operator samt två operander (tal eller strängar i form av en *Constant_node*).
eval	Returnerar resultatet av det aritmetiska uttrycket.

Klassen skapar ett aritmetiskt uttryck och beräknar det för att sedan returnera det.

Def_node

initialize(def_name, parameters, decls, return_value)	**def_name** är namnet på funktionen. **parameters** är alla parametrar som funktionen kan ta in. **decls** är alla satser som funktionen innehåller. **return_value** är vilket returvärde funktionen skall ha.
set_params(params)	Funktionen initierar alla inparametrar till ett värde som angivits vid ett funktionsanrop och lägger sedan dessa i en specifik hash som innehåller alla variabler.
delete_params	Tar bort alla parametrarna från hash:en som innehåller alla variabler.
eval(params)	Tar emot en array med värden för att sedan tilldela dom till inparametrarna i angiven ordning. Utförs bara om ett funktionsanrop har gjort (om en *Fun_call_node* har skapats). Då utförs alla satser som funktionen innehåller samt returvärdet returneras i slutet.

Den här klassen används för att skapa funktioner. Funktionen körs bara om ett funktionsanrop till den specifika funktionen görs, d.v.s. en *Fun_call_node* har skapats med samma namn och x antal inparametrar,

Fun_call_node

initialize(def_name,params)	Tar emot ett funktionsnamn och ett x antal värden (i form av *Constant_node:s*).
eval	Anropar den deklarerade funktionens *eval* metod för att denne skall evalueras.

Används för att anropa en tidigare deklarerad funktion.

Length_node

initialize(value)	Tar emot ett värde i form av en *Constant_node* eller en *Variable_node.*
eval	Beräknar längden på värdet som angavs för att sedan returnera det.

Klassen används för att beräkna längden på olika typer i vårat programspråk.

Set_image_node

initialize(img)	Tar emot en sökväg till en bild.
eval	Returnerar en ** (HTML) tag.

Används för att skriva och presentera bilder till den aktuella filen.

Array_add_node

initialize(array, value)	Som inparametrar tar den en array samt ett värde.
eval	Vid evaluering så läggs värdet till sist i arrayen.

Klassen används för att utöka arrayer.

Read_file

initialize(file)	Inparametern skall vara en sökväg till en fil.
eval	När evalueringen körs så letar klassen upp filen för att sedan läsa in den och skapa en array med *Constant_node:es*.

Används för att läsa in externa .txt filer till programmet.

Open_database

initialize(file)	Inparametern skall vara en sökväg till en fil, där filen *måste* vara strukturerad på ett specifikt sätt (se användarhandledning).
eval	Öppnar databasen och skapar en hash med alla tabeller med tillhörande information för att sedan returnera hash:en.

Klassen används för att öppna databaser i vårat programspråk.

Database_table

initialize(data, table, field = nil)	**data** är en öppnad databas (en instans av *Open_database*), eller en variabel. **table** är vilket fält som nycklarna och värdena skall hämtas från, eller vilket värde som skall skrivas ut från en angiven

	variabel. **field** (valfri) är vilket fält som skall skrivas ut.
eval	Returnerar angivet värde.

Används för att hämta ut specificerad information från en databasfil (som öppnas av en *Open_database* klass) som måste vara strukturerad på ett visst sätt.

Grammatik

Anmärkningar:

* - (*asterisk/strärna*) indikerar godtycklig många (*0..n*)
”” - betyder ett speciellt nyckelord som måste uppfyllas.

block ::= <unit> | <block>, <unit>

unit ::= "html", "(", <string>, ")", "begin", <declarations>, "end"

declarations ::= <declaration> | <declarations>, <declaration>

declaration ::= <loop> | <function> | <logic_stmt> | <assignment> | <function_call> | <function_declaration>

loop ::= "for", "(", <variable>, "in", <loop_athom>, ") do", <declarations>, "end" |
"for", "(", <variable>, "in", <loop_list> "), do" <declarations>, "end" |

loop_athom ::= <string> | <array>

variable ::= </[a-z][a-z|_]*/> (*Måste inehålla en inledande bokstav a-z följt av godtyckligt antal a-z eller understräck*)

loop_list ::= <loop_list_values>, "..", <loop_list_values>

loop_list_values ::= <variable> | <integer>

function ::= "set_img", "(", <string>, ")", ";" |
"set_img", "(", <variable>, ")", ";" |
"print", "(", <athom>, "," <color>, ")", ";" |
"print", "(", <athom>, ")", ";" |

color ::= ":", <function_name>

logic_stmt ::= <if_stmt>, <else_if_stmt>*, <else_stmt>, "end"|
<if_stmt>, <else_stmt>, "end" |
<if_stmt>, <else_if_stmt>*, "end" |
<if_stmt>, "end"

else_if_stmt ::= "elseif ", "(", <logical_expr>, ")", "then", <declarations>

else_stmt ::= "else", <declarations>

if_stmt ::= "if", "(", <logical_expr>, ")", "then", <declarations>

logical_expr ::= <logic_expr_list>, ["or" | "and"], <logic_expr_list> |
<logic_expr>, ["or" | "and"], <logic_expr_list> |
<logic_expr_list>

logic_expr_list ::= "not", "(", <logical_expr>, ")" |
<logical> |
<comp_expr> |
<variable> |
"(", <logical_expr>, ")"

comp_expr ::= <athom>, <comp_expr_list>, <athom>

comp_expr_list ::= "==" | "!=" | "<" | ">" | "<=" | ">="

assignment ::= <variable>, "=", <assignment_list>, ";"

assignment_list ::= <function_call> | <built_in_function> | <athom> | <logical_expr> | <comp_expr> | <expr>

built_in_function ::= "read", "(", [<string> | <variable>], ")" |
"open", "(", [<string> | <variable>] ")" |
<variable>, "[", [<string> | <variable>], "]", "[", [<string> | <variable>], "]" |
"len", "(", <anthom>, ")" |

function_name ::= [A-Z][A-Z|_]* (*Måste inehålla en inledande bokstav A-Z följt av godtyckligt antal A-Z eller understräck*)

function_call ::= <function_name>, "(", <items>, ")", ";"

function_declaration ::= "def", <function_name>, "(", <arguments>, ")", <declarations>, "end"

arguments ::= <variable> |
<parameters>, ",", <variable>

athom ::= <array> |
<array_index> |
<decimal> |
<integer> |
<string> |
<logical> |
<variable> |
"(", <expr> ")" |
"(", <logical_expr>, ")"

array ::= "[", <items>, "]"

items ::= <anthom> |
<items>, ",", <anthom>

array_index ::= <variable>, "[", <integer>, "]"

string ::= "[^"]*" (*Får innehålla godtyckligt antal tecken, dock inte citationstecken*)

integer ::= <nonzerodigit>* | "0"

nonzerodigit ::= "1"..."9"

digit ::= "0"..."9"

decimal ::= <nonzerodigit>*, ".", <digit>*

logical ::= "true" | "false"

expr ::= <expr>, [+|-], <term>

term ::= <term>, [*|/], <expr_list> |
<expr_list>

expr_list ::= <built_in_function> |
<array_index> |
<string> |
<decimal> |
<integer> |
<variable> |
"(", <expr>, ")"

Översiktlig beskrivning av sytemet

Vi har valt att använda kursens verktyg *rdparse.rb* eftersom att vi fann den som en bra grund att utveckla vårat språk på.

Vårat språk består totalt av åtta (8) stycken nödvändiga filer:

- configuration.rb
- functions.rb
- nodes.rb
- open_db.rb
- rdparse.rb
- run.rb
- wpg_language.rb
- style-sheet.css

Själva huvudfilen för systemet är filen *wpg_language.rb* som innehåller alla reglerna för vårat programspråk. Reglerna genererar och skapar instanser utav sk. *noder* som finns i filen *nodes.rb*.

En nod är en klass som hanterar dom olika beståndsdelarna i vårat programspråk. Det finns exempelvis en nod för alla konstanter så som *heltal, flyttal, strängar, listor o.s.v.* samt en klassnod för *logiska kontrollsatser m.m.*

Filerna *open_db.rb* och *configuration.rb* innehåller operationer för att kunna öppna *databaserna* för vårat programspråk. Där *open_db.rb* öppna databasfilen och plockar ut den väsentliga

informationen så som nycklarna och värdena och sparar ner varje *tabell* i en hash. Filen *configuration.rb* skapar variabler på det fältet som har angivits så att det sedan går att hämta ut godtycklig information.

Filen *functions.rb* innehåller några klasser för att hantera dom inbyggda funktionerna vi har sakpat, t.ex. ”noden” för att öppna en databas, där den klassen då använder sig av *open_db.rb* och *configuration.rb*.

Från filen *run.rb* så kompileras alla *.wpg* filer, där det går att ange vilken fil som skall exekveras.

Vi har implementerat alla filerna själva förutom filen *rdparse.rb*.

Språkets beståndsdelar

Vårat programspråk består huvudsakligen av parsning av en textfil. Där vi i *wpg_language.rb* har skapat sk. *tokens* som vi sedan använder för att matcha en syntax i våra regler. I reglerna så skapas det och returneras instanser av *noder* som vi samlar på oss och bygger upp till en lista (liknande ett *syntaxträd*) med den efter varandra i våran huvudregel. I slutet när hela textfilen har accepterats och parsats så stegas listan igenom och evaluerar noderna en efter en.

Presentation av tokens och syntaxträd

Vi har valt att skapa tokens högst upp filen Alla tokens är deklarerade högst upp i filen *wpg_language.rb.*

Algoritmer eller problemlösning

För att lösa problemet med prioritering i mattematiska uttryck (även logiska) så har vi inte använt oss utav *järnvägsalgoritmen* som först var ett alternativ, utan vi har då utnyttjat reglerna till detta.

Exempelvis får ett mattematiskt uttryck bestå av bla.a följande: *strängar, decimaltal, heltal o.s.v.* samt ett mattematiskt uttryck inom parenteser (prioritering). En uträkning med parenteser kommer att gå till så att uttrycket som har parenteser kring sig kommer att beräknas först och returnera en *Expr_aritm_node* (som vi har valt att kalla det), sedan kommer uttrycket utanför parenteserna att beräknas och returnera en annan Expr_aritm_node. Multiplikation och division har automatiskt prioritet före addition och subtraktion genom Ruby.

Kodstandard

Vi har eftersträvat att ha en god kodstandard i våra projektfiler där vi har indenterat alla rader på ett logiskt sätt som följer strukturen. Vi har även försökt lägga in korta kommentarer på vissa ställen som en ”sammanfattning” av ett stycke för att öka abstraktionen ytterligare ett steg.

Variabelnamn, funktionsnamn, namn på reglerna o.s.v. är något som är väl genomtänkt i projektet för att det så enkelt som möjligt skall gå att förstå vad det är för delar, exempelvis regeln för strängar har vi döpt till ”:sring” och inte nått annat orealistiskt så som ”:abc”. Vi har även vart konsekventa med hur namnen skall se ut: klasser inleds med versal följt av gemener där orden i

klassnamnet är separerade med ett understräck ”_”. Variabel-, regel- och funktionsnamn deklareras alltid med gemener och separeras med understräck mellan orden.

Användna tekniker

Vi har utvecklat vårat programspråk i programmet SciTE med tillhörande Ruby-plugin för att kunna evaluera våra filer.

Språket är uppbyggt med hjälp av *rdparse.rb* som är verktyg för att matcha olika regler med tokens.

Reflektion

Reflektion på kursen

Genom kursens gång har vi lärt oss att arbeta två stycken personer parallellt i en grupp, vilket har fungerat väldigt bra då vi har haft god kommunikation mellan varandra. Vi har även lärt oss arbeta gemensamt på distans på ett effektivt sätt, då vi bor i olika städer och inte har kunnat arbeta tillsammans i skolan varje gång.

Arbetsfördelningen har också varit bra och nya kunskaper har tillkommit då åsikter och kritik har kommit från två håll.

Vi har även lärt oss att det är viktigt att samla på sig nödvändig fakta innan det går att börja skriva och implementera saker för att undvika fel och missförstånd.

En egenskap som vi båda kommer att ta med oss är att det är viktigt att hela tiden kommunicera och informera varandra så att båda hela tiden är uppdaterade.

I början innan vi hade hunnit börja med programmeringen så tog vi oss vatten över huvudet i *användarhandledningen.* Där hade vi sagt att vi skulle implementera ett programspråk som klarade av dynamiska handlingar i samband med HTML. Vi hade några bilder över hur det skulle kunna se ut, men inte någon koll på hur vi skulle lösa det.

Efter några möten med Anders Haraldsson så kom vi fram till hur vi skulle göra vårat språk: ett programspråk som genererade statiska HTML dokument. Efter att ha ändrat oss ganska många gånger innan vi kom fram till hur vårat programspråk skulle se ut så fick vi aldrig klart nån klok *språkpecifikation,* vilket var ganska synd.

Vi hade svårt för att skriva en bra grammatik för språket i *användarhandledningen* p.g.a. alla ändringar som uppstod, även om vi hade några exempel på hur det skulle kunna se ut i kod. Men det var något som ordnade sig senare efter att implementeringen hade börjat.

Det som var svårast för oss var att komma igång med programmerandet i början, vilket resulterade i en långsam och dålig start. Detta på grund av alla ändringar i *språkspecifikationen* samt bristande kunskap om hur regler skulle skrivas i filen *rdparse.rb*. Men detta var något som löste sig med tiden.

En annan sak som var svårare än väntat var att bl.a. skapa *logiska kontrollsatser* (if-satser) i programspråket. Denna gång var det för att vi inte visste att det skulle byggas upp ett *syntaxträd* med sk. *noder* av alla beståndsdelar i programmeringsspråket, utan vi hade istället utfört alla operationer direkt i reglerna istället. Våran lösning visade sig vara omständig för att skapa *nästlade if-satser* (en logisk kontrollsats i en annan logisk kontrollsats). Detta resulterade i att vi fick skriva om språket tre gånger innan det fungerade som vi ville, då samma misstag gjordes om två gånger.

Slutresultatet av programspråket slutade med att vi har implementerat det vanligaste och det mesta vi nämnde i *språkspecifikationen* i vårat programmeringsspråk. Detta var bl.a. funktioner, funktionsanrop, kontrollsatser, slingor, aritmetiska uttryck med operationsprioritet m.m. samt

några inbyggda funktioner som kan vara till användning för att kunna vara ett HTML språk.

En punkt vi har funderat ganska mycket på är *variable scope* (att variabler som deklareras bl.a. inuti funktioner inte existerar efter att dom anropas). Vi har bara tagit hänsyn till hur variabler hanteras i funktioner, då vi tar bort dom efter ett funktionsanrop om dessa har deklarerats inuti funktionen annars låter vi dessa vara.

Vi implementerade inte heller vissa mindre funktioner som vi hade nämnt i *språkspecifikationen*, som t.ex. att det skulle gå att skriva ut information i tabeller. Men det var något vi implementerade till viss del, eftersom det går att skriva ut information som i löpande text.

Några synpunkter på programspråket ruby

- **Funktioner har alltid ett returvärde** - Kan vara jobbigt att komma ihåg att en funktion alltid returnerar något även om inget har angivits som returvärde i språket.
- **Ingen påtvingad indentering** - Är varken ett plus eller ett minus, då det kan vara skönt att bestämma själv över hur indenteringen skall ske men å andra sidan så är det inte så speciellt kul att läsa annan kod som inte är indenterad klokt då det blir svårläst.
- **Allt i ruby är objekt** - Är bra för att det enkelt går att modifiera inbyggda klasser så som strängar och heltal och forma dem så att dom passar efter behov. Dock kan det blir svårt att hålla koll på vad som är "rätt" enligt standarden i ruby om för många modifieringar på klasser sker. Varje typklass i Ruby har även sina egna funktioner och går enkelt att gå genom att skriva *klassen.funktionen(eventuella parametrar)*.
- **Hash klassen är inte sorterad** - Efter vilken ordning som nycklarna med tillhörande värden läggs till i en instans av klassen, vilket vi anser vara till nackdel då det hade varit till fördel för oss, t.ex. när vi läser in våra "databaser" så hade vi gärna sett att värdena låg i den ordningen som dom lades till.
- **Felmeddelanden** - Vi har stött på en del väldigt dåliga felmeddelanden under programmeringstiden av projektet (och även i kursen TDP007) som inte har varit så informationsrika utan då har vi vart tvungna att söka upp informationen själva för att lösa problemen.
- **Stor skalbarhet** - Vilket gör ruby till ett stort språk med många användningsområden då det är objektorienterat och även går att använda för webben samt att det är ett skriptspråk.
- **Likt python** - Är ett plus eftersom vi har lärt oss och använt python i en tidigare kurs, dock finns det några skillnader i syntaxen så som att koden *måste* indenteras i Python men inte i Ruby.
- **Plattformsoberoende** -Är ett stort plus, eftersom det även gör vårat implementerade språk plattformsoberoende.
- **Fixnum inte Integer** - I ruby är ett heltal en instans av *Fixnum* och inte *Integer* vilket det annars är i dom andra programmeringsspråken som vi har använt av oss på programmet hittills.

- **Smarta namn på metoder** - Ruby är uppbyggt på ett smart sätt angående dess egna metoder, t.ex. så går det att använda funktionen *object.has_key?(key)* för att kolla om en hash har en specifik nyckel.
- **Enkelt med regex** - Att ruby har väldigt lättanvändliga funktioner för att hantera reguljära uttryck, vilket har under lättat det många gånger i kodningen.