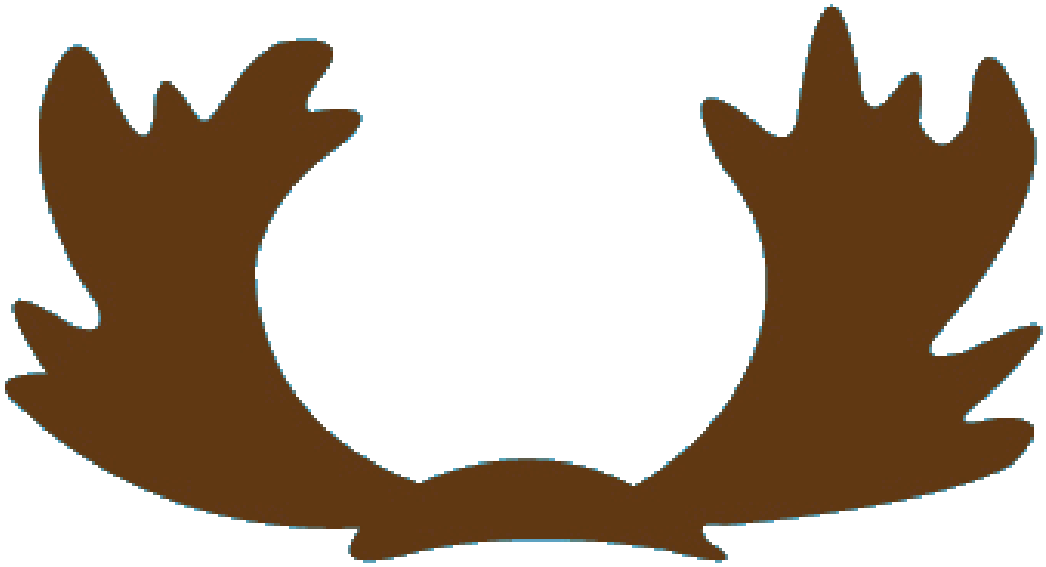

Swepp

Dokumentation av programmeringsspråket Swepp

Swepp är gjort som ett projekt för kursen

TDP019 Projekt: Datorspråk

Gustav Pettersson och Sebastian Öhrn

Innehållsförteckning

1. Innehållsförteckning 2
2. Användarhandledning 3
1. Inledning 3
2. Variabler 3
3. Matematiska uträkningar 5
4. Stränghantering 5
5. Booleska uttryck 6
6. Kommentarer 8
7. Listor 8
8. Skrivfunktionen 9
9. Om-satser 10
10. Loopar 10
11. Funktioner 11
12. Funktionsanrop 12
13. Kodexempel - Fibonacci 13

3. Systemdokumentation 14
1. Lexikalisk analys 14
2. Parsing 15
3. Evaluering 15
4. Program för arbete av projekt och kodstandard 16
5. Installeringsinstruktion 17
6. Reflektion 17

Användarhandledning

Vi har jobbat i nästan 5 månader med det här IP-projektet. Målet med projektet var att bygga ett programmeringspråk. Språket som vi gjort har vi döpt till "Swepp", som är tänkt att vara likt programmeringsspråket "C++" i en del delar och är tänkt att vara för nybörjare och användare med lite erfarenheter av programmering. Vi har en svensk syntax ("nyckelord" som används när man skriver koden för språket) och det var därifrån vi fick namnet "Swe" ifrån. Delen "pp" fick vi ifrån C++:s två plus på slutet.
Att språket är på svenska gör det lättare för dom som programmerar och inte kan engelska så bra eller överhuvudtaget. Vi hoppas att ni som användare kommer få en bra uppfattning vad grunderna i programmering efter användning av vårt språk. Ni kanske t.o.m. våga dig på ett nytt programmeringsspråk efter ni testat Swepp.

Inledning

Swepp är ett nybörjarspråk som är till för den som vill lära sig den grundläggande strukturen för programmering. I och med att språket har svenska nyckelord så passar det även mycket bra till användare som har få eller inga engelska språkkunskaper.

Det som är specifikt med vårt språk Swepp är att varje program börjar med ett "start" och ett "slut". Där emellan har man allt som programmet ska innehålla, vilket exempelvis kan vara funktioner och tilldelning av variabler som heltal, decimaltal, textsträngar, booleska uttryck och listor. Listor kan i sig kan innehålla heltal, decimaltal och textsträngar. I ett program kan man även göra loopar i form av en "för"-loop och "medans"-loop. Språket Swepp har även kontrollsatser kallade "om", "annars_om" och "annars" som man kan använda för att kontrollera om ett booleskt uttryck evaluerar sant eller falskt och utifrån det köra olika satser. Allt det vi nämnt innan kan man även göra i en funktion som vi sedan kan kalla på med hjälp av funktionsanrop med olika värden till dess parametrar.

Variabler

Variabler kan ses som lådor som man sparar värden i. I vårt språk så måste man deklarera (skapa) variablerna innan man kan använda dom. Vi har även gjort ett krav att man deklarerar variablerna en viss typ när man skapar dom, och endast den typen av variabler kan sparas ner som värde.

Om man till exempel skriver:

```
tal x = 3
```

så deklareras variabeln x till 3.

Man kan inte byta vilken typ en variabel är, men man kan byta värdet på den, så länge den får ett värde av samma typ. Detta gör man på följade sätt:

```
x = 7
```

Detta fungerar endast om man redan har deklarerat x (som vi gjorde innan), annars får man ett felmeddelande.

```
x = "hej"
```

Detta skulle inte fungera eftersom "hej" är en sträng.

Variabler kan vara av typerna "sträng", "tal", "lista" och "bool" (sanningsvärde). Variabelns namn består av en bokstav, stor eller liten, som kan vara följd av: fler bokstäver, siffror, punkt eller understreck i vilken ordning som helst (så länge man börjar med en bokstav). Så man kan döpa en variabel till t.ex.:

```
x
X
hästar
H3574r
t_3fa3._d.
```

men man kan inte döpa variabler till följande:

```
3as
(parantes
hus-färg
_test_
```

Som i exemplen på riktiga variabelnamn så kan man döpa en variabel till litet "x" och stort "X". Det kommer inte räknas som samma variabel, så man kan tilldela x till 4 och X till "hej" genom att skriva:

```
tal x = 4
sträng X = "hej"
```

Man kan även göra båda till samma typ av variabel, så att X är något tal istället för en sträng.

Matematiska uträkningar

Om vi ska göra ett enkelt program där vi gör en matematiskt uträkning med variabler så kan det se ut så här:

```
start
   tal äpplen = 10
   tal korgar = 2
   tal äpplen_per_korg = äpplen / korgar
slut
```

De giltiga matematiska operatorer i språket är:

```
+ - / * %
```

Tecknet % är resten vid division av två heltal. Om man skulle skriva 10%3 så skulle det få resultatet 1. Man kan även använda sig av parenteser för att göra uträkningar och i och med det få den prioritet man vill mellan sina operatorer. Exemplet nedan förklarar skillnaden i resultat man kan få.

```
2*(3+4) = 14
2*3+4 = 10
```

För att göra en tilldelning av "tal" (som kan vara både decimaltal och heltal) så skriver man som ovan nyckelordet "tal" följt av ett variabel namn, t.ex. "äpplen" och ett lika-med tecken "=" och ett giltigt värde vilket i detta fall är ett heltal eller decimaltal då nyckelordet är "tal".

Stränghantering

För att göra en tilldelning av "sträng" som är en icke formaterad text mellan två "-tecken eller '-tecken, gör man likt all annan tilldelning i vårt språk, vilket är nyckelord följt av variabelnamn, sedan ett lika-med tecken följt av värdet som i detta fall är en sträng. Ett exempel på tilldelning av strängar, samt addering av strängar kan se ut på följade vis:

```
start
     sträng förnamn = "Bosse"
     sträng efternamn = 'Larssson'
     sträng fulltnamn = förnamn & " " & efternamn
slut
```

Med hjälp av operatorn "&" så kan man sätta ihop flera strängar med varandra. Variabeln fullnamn kommer nu få värdet "Bosse Larsson". Notera att det går att kombinera addering av strängar med strängar som inte är bundna till en variabel. I vårt fall så gör det att vi kan få ett mellanslag emellan förnamnet och efternamnet så fullnamn blir formerat som vi vill.

Booleska uttryck

I Swepp kan man även kontollera om ett påstående med tal eller variabler med värden av tal är sant eller falskt. Vill vi t.ex. kontrollera om variabeln a är mindre än 10 så kan vi skriva på följade vis:

```
a < 10
```

Man kan även göra större uttryck genom att använda parenteser.

```
((a < 4) och (a<3)) och (a>1)
```

Här är en lista över giltiga booleska operatorer i vårt språk:

mindre än: <
mindre eller lika med: <=
större än: >
större eller lika med: >=
lika med: ==
inte lika med: !=

Booleska uttryck kan vi använda för att kontrollera hur länge en loop ska köras och om en "om"-sats, "annars_om"-sats eller "annars"-sats ska köras. Vi kan även skriva ut ett booleskt uttryck eller tilldela det till en variabel. Om vi ska tilldela en variabel ett boolesk uttryck så kan det se ut så här:

```
bool a = 10 < 99
```

Detta skulle ge variabeln a värdet sant som vi sedan skulle kunna jämföra med ett annat booleskt uttryck.

Om man vill jämföra två olika booleska uttryck så finns det några giltiga operator som stöder det. Dessa är:
"och", "inte", "eller", "antingen_eller".

Nyckelordet "och" används för att kontrollera om två stycken booleska uttryck är sanna. Om båda booleska uttrycken genererar sant så blir även "och" sant. Men om bara en eller ingen av de booleska uttrycken är sanna så blir "och" falskt.

Denna generar sant:

```
1 < 10 och 9 > 8
```

Detta generar falskt:

```
2 < 10 och 7 >= 8
```

Detta generar falskt:

```
1 > 10 och 8 != 8
```

Nyckelordet "inte" gör så att man får ut det motsatta värdet utav ett booleskt uttryck.

Detta generar sant:

```
inte 2 <= 1
```

Detta generar falskt:

```
inte 10 >= 9
```

Nyckelordet "eller" kräver bara att ett av de två booleska uttrycken man jämför med är sanna men det blir även sant och båda uttrycken skulle vara sanna:

Detta generar sant:

```
1 > 10 eller 1 < 10
```

Detta generar sant:

```
1 <= 3 eller 2 >= 1
```

Detta generar falskt

```
10 > 20 eller 20 == 13
```

Nyckelordet "antingen_eller" returnerar sant om enbart ena satsen är sann, och den andra är falsk:

Detta generar sant:

```
1 > 2 antingen_eller 3 < 4
```

Detta genererar falskt:

```
2 >1 antingen_eller 4 > 3
```

Detta generar falskt:

```
2 < 1 antingen_eller 4 < 3
```

Kommentarer

Kommentarer i koden är endast till för programmerarna att läsa och kommer inte att köras. Man kan skriva kommentarer i koden genom att skriva // på en rad, så kommer resten av raden att ignoreras när koden körs. Man kan även använda sig av /* och */ för att ha "flerradskommentarer". "Flerradskommentarer" behöver inte vara på flera rader, utan kan användas mitt i koden.

Alla dessa kommentarer är giltiga i vårt språk Swepp

```
om ( /*första siffran här -> */ 1 < 2)// en enrads kommentar
/* här är
en flerrads
kommentar */
   skriv "hallå!"
slut
// en till kommentar
```

Listor

Om en variabel ses som en låda, så kan en lista ses som en hylla, där flera värden kan sparas i samma variabelnamn. Man kan spara ner vilka värden man vill i en lista. Första värdet i en lista har index 0, andra har 1 osv. Så om man vill ta bort femte värdet så får man skriva 4. Om man vill ha ut sista elementet i listan kan man skriva -1. Om man vill ha det näst sista värdet så skriver man -2 osv.

Programexemplet nedan visar alla operatorer man kan göra med listor i Swepp.

```
start
     lista listan = [1,54,"kalle",sant]
     listan lägg_till "anka"
     skriv listan                 // [1,54,"kalle",sant,"anka"] skrivs ut
     listan ta_bort_index 3       // tar bort sant från listan
     listan ta_bort_värde "kalle" // tar bort alla element med värdet
                                  // "kalle" från listan
     skriv listan[1]              // skriver ut 54
     tal antal = storlek_på listan// antal får värdet 3, eftersom det är nu
                                  // 3 värden i listan
slut
```

Den hylla som vi beskrev listan som innan är som exemplet ovan, två hakparateser och innehållet läggs i hyllan med ett komma emellan för att avskilja. Vi kan med hjälp indexering ta ut ett specifikt innehåll i listan. Det gör vi genom att ange namnet på listan följt av hakparenteser och inne i hakparenteserna så skriver vi var i listan elementet vi är ute efter ligger med en början på 0.

Skrivfunktionen

För att få ut innehållet av variabler eller få ut returvärdet av en funktion direkt på skärmen när man kör programmet så kan man använda sig av den inbyggda skrivfunktionen. För att använda den skriver man helt enkelt nyckelordet "skriv" följt av en datatyp eller variabel. Detta kan se ut på följade sätt:

```
sträng långt_ord = "flagstångsknopp"
tal pi = 3.14
skriv "bosse é bäst!"
skriv långt_ord
skriv pi
```

Det fungerar även att skriva ut allt som kan tilldelas, d.v.s. det går att skriva ut matematiska uträkningar, booleska uttryck och strängaddering på samma sätt som man skriver ut tal, strängar och variabler.
Exemplet nedan bekräftar detta:

```
skriv "bosse é bäst!" & " och lars äter skor"
skriv (1 < 99) och (15 <= 10)
skriv (99*101-55) - (16*55-3)
```

Om-satser

Om-satser är en sats som kontrollerar ett eller flera villkor och kör olika stycken med kod beroende på vilka som är sanna. Om-satserna kan grupperas ihop med "annars_om" och "annars"-satserna. En vanlig om-sats består av texten "om" och ett boolesk uttryck inuti en parantes. Till exempel:

```
om(1<2)
   skriv "ett är mindre än två"
slut
```

"annars_om" och "annars" kan också läggas till och man kan ha inga eller fler annars_om, men bara noll eller en annars-sats. "annars-satsen" kommer alltid sist. Om den vanliga "om-satsen" är sann så kommer inte "annars_om" satserna eller "annars" satsen köras.

```
om(3>5)
   skriv "tre är större än 5"
annars_om(7<19)
   skriv "7 är mindre än 19"
annars_om(6>2)
   skriv "sex är större än 2"
annars
   skriv "inget blev sant"
slut
```

I detta exemplet så kommer "7 är mindre än 19" skrivas ut, men inte "tre större än 5" eller "sex är större än 2". Om varken om-satsen eller annars_om satserna hade varit sanna så hade "inget blev sant" skrivits ut

Loopar

En loop är i vårt språk en stycke programkod som körs så länge som loopens booleska uttryck är sant. I vårt språk har vi en "för"-loop och en "medans"-loop som fungerar ungefär på samma sätt men med några olikheter.
En "medans"-loop kan se ut på följade vis:

```
start
    tal gånger= 1
    tal fem = 5
    medans( gånger <= 10 )
        skriv fem*gånger
        gånger = gånger +1
    slut
slut
```

Vi börjar vår loop med nyckelordet "medans" följt av parenteser som vi fyller med ett booleskt uttryck. I exemplet ovan har vi ett booleskt uttryck som är sant så länge variabeln "gånger"s värde är mindre än eller lika med 10. Dvs. så länge det booleska uttrycket är sant så körs loopen om.
I exemplet ovan så skriver vi ut femmans multiplikationstabell upp till 5*10 med hjälp av vår "medans"-loop.

En "för"-loop kan se ut på följade vis i vårt språk:

```
start
    tal gånger= 0
    tal fem = 5
    för(gånger < 10 ; gånger = gånger + 2 )
        skriv fem*gånger
    slut
slut
```

Skillnaden på en "medans" och en "för"-loop i vårt språk är att man kan göra tilldelning inuti "för"-loopens parenteser. Detta har vi valt att göra eftersom ge de som känner igen denna slags konstruktion av loopar från andra språk, t.ex. språket C++ lite extra glädje. En annan anledning är att man då kan visa mer tydligt vad som kommer ändras för varje gång och i detta fall hur det påverkar det booleska uttrycket för varje varv i loopen.

Funktioner

Om vi nu skulle vilja göra vår uträkning i en funktion så kan vi skriva följande:

```
start
    tal funktion äpplen_och_korgar(a,b)
        tal c  = a / b
        returnera c
    slut
    äpplen_och_korgar(10 , 2)
slut
```

Som vanligt börjar vårt program med ett start och ett slut. Därefter har vi våran funktion där vi först skriver returvärdet vi har tänkt att funktionen ska ha. De giltiga returvärden som finns är "tal", "sträng", "lista" och "inget". Sedan skriver vi nyckelordet "funktion" följt av vårt funktionsnamn som i detta fall är "äpplen_och_korgar". Sedan skriver vi inom två parenteser namnet på våra parametrar vilket kommer få värden när vi sedan gör ett funktionsanrop.

I en funktion kan vi sedan ha alla våra giltiga kodsatser som tilldelning och loopar. För alla funktioner förutom "inget" funktioner så måste man ha ett returvärde vilket man ger genom att ange nyckelordet "returnera" följt av en variabel eller datatyp av samma typ som funktionen.

En funktion måste likt programmet anslutas med nyckelordet "slut".

Funktionsanrop

Det som är så speciellt med en funktion är att en funktion kan återanvändas och få ut olika värden men ändå gör exakt samma arbete som innan men med olika värden som den tar in. Funktionsanropet är det som lyder "äpplen_och_korgar(10 , 2)" i kodsnutten ovan. Man skriver först namnet på funktionen man vill kalla på och sedan så skriver man inom parenteserna vilket värde man vill ge funktionens parametrar. I detta fall får parametern a värdet 10 och parametern b värdet 2 och vi får ut returvärdet 5 av vår funktion. Om vi skulle skicka in värdena 100 och 10 till vår funktion så skulle vi få returvärdet 10 istället. Eftersom funktionsanrop för alla funktoner utom "inget"-funktioner ger ett returvärde så går det bra att tilldela en variabel returvärdet från funktionsanropet.

```
tal returvärde = äpplen_och_korgar(10 , 2)
```

Variabeln "returvärde" får i exemplet ovan värdet 5.

Kodexempel - Fibonacci

Här är ett kodexempel för att visa att det går att göra ganska avancerade uträkningar i språket Swepp om man skulle vilja gör det. Exemplet nedan har en funktion för att räkna ut vad ett tal skulle bli i Fibonacci-serien.

```
start
    tal funktion fib(n)
        om(n<2)
            returnera n
        slut
        lista fibbo = [0,1]
        tal count = 1
        medans(count < n)
            count = count + 1
            fibbo lägg_till fibbo[-1]+fibbo[-2]
        slut
        returnera fibbo[-1]
    slut
    tal fib_tal = fib(10)
    skriv fib_tal
slut
```

Systemdokumentation

Vårt språk Swepp är byggt på lexern rdpase och utifrån det så har vi byggt vårt eget språk med hjälp av exempel från språket diceparser och filen syntaxträd.rb som fanns tillgänglig på kurshemsidan. Filen bestod av exempel på noder. Språket vi har gjort är ett nybörjarspråk med svenska nyckelord för att göra det mer förståligt för de som har lite eller ingen kunskap i det engelska språket som i vanliga fall är standarden för programmeringsspråk. Swepp gör det enkelt för användaren att förstå grunderna i programmering men lär även användaren att förstå skillnaden på de olika datatyperna vid t.ex. deklaration av variabler.

Lexikalisk analys

Eftersom vi har använt oss av lexern rdparse så har vår lexikaliska del endast bestått av tokens som i sig består av reguljärauttryck i språket Ruby. Med hjälp av dessa tokens så kan vi ifrån en sträng ta ut det vi vill och matcha det i våra regler i vår parser. De olika token vi har är följade och är även matchade i ordningen nedan:

Här matchas flerradskommentarer och returnerar inget vilket gör att de fungerar som de ska göra, dvs. de ignorerar allt inom en enradskommentar.

```
token(/\/\*(.|\n)*\*\//)
```

Här matchas flerradskommentarer och returnerar inget av samma anledning som flerradskommentarens token.

```
token(/\/\/(.)*$/)
```

Här matchas blanksteg, eftersom de likt kommentarer ska ignoreras så returnerar vi inget.

```
token(/\s+/)
```

Dessa tokens matchar våra tal i språket, vilket kan vara negativa och positiva heltal och decimaltal. Med hjälp av matchningen så tar vi ut talen i form av strängar och returnerar dom som Integers och Floattal beroende på om de är heltal eller decimaltal.

```
token(/-(\d+)/) {|m| m.to_i}
token(/-(\d+[.]\d+)/) {|m|m.to_f}
token(/\d+[.]\d+/) {|m| m.to_f}
token(/\d+/) {|m| m.to_i }
```

Dessa två tokens matchar det som vi kommer använda som strängar i vårt språk.

```
token(/"[^\"]*"/) {|m| m }
token(/'[^\']*'/) {|m| m }
```

Den första token matchar strängar där innehållet skrivs inom citattecken likt:

"Detta är en sträng".
Den andra matchar strängar som där innehållet skrivs inom apostroftecken likt:
'Detta är också en sträng'.
I citat-teckens strängar så kan man använda apostrof-tecken utan att avsluta strängen och i apostrof-teckens strängar så kan man använda sig av citat-tecken. Vi hade först tänkt att ha ett specialtecken \" som skulle göra att man skulle kunna ha citat-tecken i en vanlig sträng men vi kom inte på något bra sätt för det och bestämde att man får helt enkelt göra en annan struktur på sin sträng.

Våra sträng-tokens returnerar bara sig själva eftersom allt är strängar från början då det är en textfil man skriver sitt språk i som sedan görs om vid behov.

Denna token tar ut alla olika specialtecken vi använder i vårt språk. Här matchats t.ex. matematiska operatorer, tecken som används vid booleska jämförelser osv. Här returnerar vi bara dom tecknen som matchas.

```
    token(/(\+|\-|\*|\/
|!=|\.|%|&|\(|\)|\[|\]|\:|;|<=|!|>=|<|>|==|=|\,|<<,>>)/) {|m| m}
```

Här så matchar vi vad som kommer bli variabler i vårt språk. Variabler måste börja på en stor eller liten bokstav från A till Ö följt av godtyckligt många stora eller små bokstäver från A till Ö eller av siffror från 0 till 9 eller understreckstecken. Även variabler returnerar matchen i sig.

```
    token(/[a-zA-ZåäöÅÄÖ]+[a-zA-ZåäöÅÄÖ0-9_]*/) {|m| m}
```

Parsing

På det sättet vi har gjort vårt språk Swepp, så består vår parsning av en stort antal regler där vi kan matcha vad reglerna ska stå för och sedan bestämma av som ska hända om en regel matchas. Vår grundregel är vår program regel som säger vad ett program får bestå utav. Denna regel säger att ett program måste bestå av nyckelordet "start" följt av vad regeln statements säger är giltigt och sedan avslutas program reglen med nyckelordet "slut". Regeln statements får bestå av ett statement eller statements följt av statement. Statement reglen matchar alla andra regler som säger vad som är giltigt att göra i vårt program. Detta kan t.ex. vara tilldelning av variabler, loopar, listhantering o.s.v. Med hjälp av denna struktur så är vårt program rekursivt och fungera som det ska. Alla våra regler som matchas i statment returnerar en nod som står för evalueringen och vår statement regel returnerar det returvärde den fick ifrån regeln i en ruby-array. I regeln statements sätts sedan alla dessa ruby-arrayer ihop och när programmet är färdigt så ligger hela programmet i form av noder i rätt ordning i en enda stor array som sedan itereras och kör nodernas uträkningsfunktion.

Evaluering

Vår evaluering sker som nämndt innan i form av noder. Dessa noder är alla olika klasser och består utav en initieringsfunktion och en evalueringsfunktion. I de allra flesta fall så sparar

initieringsfunktionen ner de värden den får i instansvariabler. Det kan även ske olika kontroller för att se att man har fått in rätt slags värden, men i slutändan så sparar initieringsfunktionen värden till instansvariabler. Noden evalueringsfunktion gör all uträkning och returnerar sedan ut värden som man förväntar sig vid t.ex. en addition av två tal. Det är även här som vi gör de flesta kontrollerna som t.ex. att man returnerar rätt typ när man gör en funktion eller att man inte försöker skicka in för många eller få parametrar vid et funktionsanrop än det giltiga. I evalueringsfilen som vi har vår hash-tabell med alla värden som vi sparar ner i variabler. Vi har även våra funktioner i en speciell hash-tabell där namnet på funktionen är nyckeln och själva funktionsnoden är värdet. På det sättet kan vi även kontrollera att man inte får kalla på en funktion som inte finns och att man inte får kalla olika funktioner samma namn osv. Vi hade tänkt oss i början att vi skulle lyckas göra nästlade funktoner och variabler, men vi lyckades inte komma fram till hur vi skulle göra det och vi fokuserade istället på att implementera allt annat vi skulle ha i språket. Vi sparar ner allt globalt istället vilket fungerar bra men gör att man inte kan få funktioner rekursiva t.ex. Vi skulle även velat ha ett smidigt sätt att kontrollera att en variabel av t.ex. tal får ett värde av rätt typ. Vi löste det så att vi skapade en lista för varje datatyp som innehåller namnet på variabeln och på så sätt kan vi kontrollera att variabler som t.ex. finns i vår stränglista vara får ha värdet av strängar. Det som blir jobbigt är att det blir så pass många kontrolleringar men eftersom vi bara har ett fåtal olika datatyper så fungerade det acceptabelt.

Program för arbete av projekt och kodstandard

Vi började skriva vårt språk i programmet emacs som gjorde det ganska enkelt att få en snygg och rätt indenterad rad, men vi gillade inte alla udda kortkommandon som fanns och vi tyckte det var onödigt svårt och komplicerat att för att kopiera, klistra in kod, indentera en helt block av kod, söka och ersätta kod osv. Vi gjorde istället som många andra i klassen, vi började skriva språket i programmet SciTE istället som har många och igenkännbara kortkommandon och fungerar smidigt att skriva i. Det som vi fick problem med var indenteringen som inte ville indenteras som vi ville och det blev även konstigt då man kopierade och klistrade in kod. När vi till sist blev klara så använde vi oss av programmet NetBeans som andra i klassen hade sagt hade en funkton för att få sin kod rätt indenterad.

Vi har inte följt någon känd kodstandard för namngivning av variabler eller för indentering utan vi har försökt göra koden snygg enligt vår egen standard, med engelska variabler som inte får innehålla stora bokstäven förutom om de är klassnamn och vi har understäck istället för mellanslag då Ruby inte förstår variabeln med mellanslag utan tolkar det som två olika variabler. Vi har försökt få så självförklarade variabelnamn som möjligt men ibland så är det svårt att komma på unika variabler, speciellt då man bara ska iterera en lista med noder och köra dess evalueringsfunktion.

Installeringsinstruktion

För att man ska kunna köra Swepp kod så måste man ha Ruby installerat på datorn. På Windows datorer så kan man ladda hem Ruby från `http://rubyinstaller.org/` och sedan köra den nerladdade filen. På Linux datorer så kan man ladda hem och installera Ruby genom att skriva `sudo apt-get install ruby1.9.1-full` i terminal fönstret.
Projektet ligger nerpackat i en zip-fil som man kan hämta ifrån `http://uploading.com/files/57d841ac/SWEPP_FINAL.zip/`. I ubuntu så är det bara att högerklicka på zip-filen och välj "öppna med Arkivhanteraren". Där ifrån kan du sedan välja var du vill packetera upp projektet. Projektet kommer ligga direkt i mappen där filen du skriver i kommer heta "swepp_01.swepp". För att sedan köra och skriva i Swepp så gör det lättast i programmet SCiTE som går att ladda ner från hemsidan `http://www.scintilla.org/SciTE.html`. De två filer som behöver öppnas med SCiTE är swepp_01.rb och swepp_01.swepp. Det är swepp_01.swepp som du som användare kommer skriva din kod i och för att sedan köra programmet så ändrar du till filen swepp_01.rb och tycker bara F5-knappen på ditt tangentbort så körs ditt program.

Reflektion

Vi utgick från rdparser uppgiften i TDP007 och började jobba med att ändra den för att få som vi ville. Vi hade rätt stora problem att förstå hur det fungerade över huvud taget, men när vi väl kom igång så gick det rätt bra. Tills vi fick reda på att man skulle använda noder. Det gjorde att vi fick göra om mycket av det vi redan gjort. Vi hade ingen direkt aning om vad som menades att man skulle använda sig av noder, eller hur dom fungerade. Men det visade sig att det var klasser som man sparade ner data i, som man senare "körde" när man anropade klassens "seval"-funktion. Att man var tvungen att använda noder var för att t.ex. if-satser skulle fungera som dom skulle, så att inte allt i if-satserna kördes oavsett vad som stod i villkorssatsen.

Vi valde att ha svenska nyckelord i vår syntax, som är bl.a. ovant, men även lite jobbigare att skriva, då "för"-loopar inte låter i närheten av lika naturligt som en "for"-loop. Detta märks mycket bättre på "elsif" som vi gjorde till "annars_eller". Det finns även en del ord som låter konstigt eller är svårt att översätta till svenska bra, så att man fortfarande förstår vad som menas.

Våra variabler har vi gjort så att man sätter dom till en viss typ, som i C++. Detta skulle vi kunnat strunta i, då det i princip bara ger oss mer jobb och att det blev rätt mycket mer att skriva/felkontrollera. Vi hade från början tänkt att man kunde ha alla bokstäver, siffror, understreck och bindestreck i variabel namn. Men vi tog bort bindestrecket för att vi kom fram till att det borde kunna skapa problem vid subtraktion med variabler.
Vi tänkte ha stränghantering i vårat språk, t.ex.:
sträng a = "hej"
sträng b = a[1]

För att spara ner "e" i variabeln b. Men vi hade inte tid till detta så vi har inte implementerat det.

Arrayer gjorde vi också i C++ stil, så att man har "tallista" och "stränglista" o.s.v. Vi hade lite problem med arrayer. Vi lyckades spara ner dom på första försöket, men vi kunde bland annat inte få ut värden i variablerna. T.ex.:
skriv var[3]

När vi väl lyckades göra detta så var det inte speciellt svårt att lägga till så att man kunde ta bort och lägga till element i arrayen. Vi kan även ta bort värden med index nummer eller värde.

Funktioner trodde vi skulle vara svårare än det var, men det var i princip bara att spara ner alla parametrar och satser i en nod, och sedan köra funktionens "seval"-funktion vid funktions anrop.
Vi har inte implementerat att man att det ska vara en speciell variabel typ i parametrar, så man kan skicka in vad man vill för tillfället. Man kan skriva in default värden när man skapar klasser, men har inte haft tid att implementera att så de kan användas, då man måste skicka med alla parametrar när man anropar klasser. Våran returnera funktion kan enbart användas i funktioner, och kan enbart returnera ett värde med samma typ som man deklarerade funktionen med. Man kan inte returnera något värde med en "inget funktion", och man kan enbart returnera ett tal med en "tal funktion". Vi kom inte på hur man skulle skapa ett krav på att man har ett retur-anrop i en klass, så man behöver inte returnera något i någon funktion.

Vanliga if-satser hade vi inte något större problem med, men när man hade med elsif och else så visste vi inte hur vi skulle göra för att den skulle "stanna" när den hittade en sats som blev sann. Vi testade lite olika sätt att få det att fungera bra, men vi lyckades inte komma på något. Vi fick hjälp med hur vi skulle lösa denna och det gjorde vi genom att lägga alla if-satserna i en lista, och loopa igenom tills vi hittade en som var sann. Vi satte else-satsen så att den alltid blir sann, så att den alltid körs om ingen if-sats eller elsif-sats var sann.

While-loopar var lätta att göra efter vi hade gjort if-sater. Det var bara att göra en koll om den var sann, och köra koden tills den inte var det längre.

For loopen var lite jobbigare att göra, men inte mycket, då vi gjorde for-looparna rätt lika for-loopar för C++:
för(i<2;i=i+1)

Så det var bara att köra andra delen efter var gång som satserna i for-loopen kördes.

Break och returnera satserna hade vi lite problem med när vi hade dom i loopar och i if-satser, då det inte skickades tillbaka riktigt. Det fungerade bra när man inte använde dom i loopar eller if-satser. Vi var tvungna att kolla i looparna och if-noderna om man fick med ett return/break nod, och då returnera värdet och bryta körningen av resten i koden i loopen.

För att summera det hela så hade vi en hel del problem i början men när vi började få koll på hur allt skulle vara strukturerat så gick det lättre. Vi tycker att vi hade fått med det mesta i språket som vi i början hade tänkt att det skulle innehålla. Det som vi saknar dock är variabel och funktionsnestning, vilket vi tycker att ett riktigt språk borde ha men det var något vi inte hann med helt enkelt.