Seraphina

Jag väljer dig!

Gustav Bylund & Alexander Kazen
Tekniska högskolan vid Linköpings universitet
2013

Innehållsförteckning

Inledning

Vi har läst kursen TDP019 Projekt: Datorspråk vid Institutionen för datavetenskap på Tekniska högskolan vid Linköpings universitet. Huvuddelen av denna kurs bestod av ett projekt med målet att skapa ett datorspråk. Vi har valt att skapa språket Seraphina som presenteras i detta dokument.

Bakgrund

Idén till Seraphina växte fram under en lunch tillsammans på universitetet. Vi tyckte att det skulle vara bra att lära barn programmering tidigare, då det är något som man kan ha nytta av även ganska tidigt i livet och som även går att applicera på andra områden. Men de flesta programmeringsspråk är idag relativt komplicerade och inte särskilt intuitiva. Det är svårt för någon som precis lärt sig läsa att förstå vad olika ord och konstruktioner ska betyda. Ofta finns det många specialtecken, som används för att till exempel markera upp koden i block, och för någon som inte är insatt är det svårt att förstå vad som betyder vad. De flesta stora språk som finns idag är dessutom skrivna på engelska, vilket är ännu ett hinder för någon som är uppväxt i Sverige och kanske inte hunnit lära sig engelska bra. Om man slapp den språkliga barriären skulle det vara lättare att motivera nybörjare att lära sig programmera.

Det här dokumentet är riktat till dig som förälder, syskon, lärare eller något annat som gör att du kan tänkas vilja hjälpa ett barn att komma igång med programmering.

Syfte

Syftet med Seraphina är alltså därmed att skapa ett språk som är dels på svenska, dels så grammatiskt likt svenska i sina konstruktioner att framförallt barn, men även andra nybörjare som aldrig programmerat, lätt ska kunna följa vad som händer i olika delar av skrivna program. Språket är inte menat att användas för avancerade programkonstruktioner eller för att skapa kraftfulla verktyg, utan snarare som ett insteg till programmering i allmänhet, riktat till dem som aldrig skrivit en rad kod tidigare.

Användarhandledning

Nedan följer en sammanfattande beskrivning av språket Seraphina. Handledningen innehåller all information som behövs för att installera Seraphina från grunden och skriva ett första program. Den innehåller även beskrivningar av språkets alla konstruktioner och exempelkod som kan hjälpa användaren att förstå hur man bör tänka när man utvecklar program i Seraphina.

Installation

För att kunna skapa och köra program i Seraphina behövs en valfri texteditor, ett system med Ruby installerat och slutligen filerna för Seraphina. Ruby går att hitta på: http://www.ruby-lang.org/en/downloads/ och är gratis att ladda ned och använda.

Seraphina finns att hämta på https://github.com/Maistho/Seraphina.

För att installera Seraphina packas filen `seraphina.tar.gz upp`, sedan körs den installationsfil vars filnamn matchar operativsystemet som används. För att avinstallera Seraphina kör man enklast uninstall-filen som ligger i samma mapp.

Komma igång

Seraphina är ett interpreterat språk och det finns därför två olika sätt att exekvera kod. Antingen sparas koden som ska köras i en fil (Seraphina använder sig av filändelsen `.sp`), vilket är en fördel om man vill skriva ett program som ska användas flera gånger, eller så skrivs koden direkt i en interaktiv klient.

För att starta den interaktiva klienten kör man programmet `seraphina` utan några parametrar från kommandoraden. Om man istället vill köra ett program skrivet och sparat i en .sp-fil, till exempel filen `hej_världen.sp`, körs programmet med filnamnet som parameter. Om man skriver filändelsen eller inte spelar ingen roll. Med andra ord, för exemplet `hej_världen.sp` går det att köra antingen

```
seraphina hej_världen
```

eller

```
seraphina hej_världen.sp
```

"Hej världen!" och Seraphina från grunden

Som en introduktion till Seraphina tar vi här upp det klassiska "Hello world!", här i svenskt format, och visar hur det skrivs i just Seraphina. Koden för programmet är simpel:

```
skriv "Hej världen!"
```

Denna rad kan antingen köras direkt i den interaktiva klienten eller sparas i en fil med filändelsen `.sp` för att köras om och om igen.

Nedan följer en genomgång av alla nyckelord och konstruktioner i Seraphina.

Datatyper
I Seraphina finns det tre olika datatyper, heltal, decimaltal och text.

Ett `heltal` kan till exempel vara 1, 2, 10 eller -3 och skrivs i koden som siffror utan några extra tecken. Interpretatorn känner igen siffror på denna form som `heltal`.

Ett `decimaltal` kan till exempel vara 0,4, -1,3 eller 3,14, och skrivs även det direkt i koden utan några specialtecken runt. Interpretatorn känner igen ett tal på formen siffra/siffror följt av ett kommatecken följt av siffra/siffror som ett `decimaltal`.

Slutligen finns datatypen `text`, som till exempel kan vara "hej", "fjäril" eller "programmerare". För att tala om för interpretatorn att datatypen är `text` sätter man citattecken runt texten man vill skriva.

Tilldelning och variabler
För att spara data och kunna använda den senare behöver datan lagras i en variabel. För att tilldela en variabel ett värde används nyckelordet `blir`. Eftersom Seraphina använder sig av lös typning så är det enkelt att skapa en ny variabel och tilldela den ett värde. För att skapa en variabel `a` och tilldela den värdet `3` gör man på följande sätt:

```
a blir 3
```

Det går även att tilldela en variabel värdet från en annan variabel:

```
a blir "alfabet"
b blir a
```

Nu kommer `b` ha samma värde som `a`. Värt att notera är att `b` får det värdet som `a` hade vid tilldelningen, och att om värdet på `a` ändras kommer inte värdet på `b` ändras.

In- och utdata

I Seraphina finns två nyckelord kopplade till in- och utmatning av data. Det första är `skriv` som visades ovan, vilket används för att skriva ut data på skärmen. För att göra en utskrift av data använder man nyckelordet `skriv` följt av den data som ska visas på skärmen. Denna data kan vara antingen data i klartext eller data lagrad i en variabel. För att skilja på olika typer av data används de konstruktioner som nämns i avsnittet om datatyper. För att skriva ut data ur en variabel skrivs bara variabelnamnet direkt efter `skriv`. Koden kan med andra ord se ut på följande sätt:

```
tidsenhet blir " år"
skriv "Jag är "
skriv 21
skriv tidsenhet
skriv " gammal."
skriv nyrad
```

Vilket kommer ge utskriften:

```
Jag är 21 år gammal.
```

I exemplet används även `nyrad`. Detta är en fördefinierad konstant som innehåller texten "\n", det vill säga escapesekvensen för en ny rad. Självklart kan man skriva "\n" direkt, men konstanten finns fördefinierad för att underlätta och för att göra koden mer läslig för målgruppen.

Det andra nyckelordet i sammanhanget gäller inmatning av data. I många program kan inmatning från användaren vara av intresse. I de fall man vill låta användaren mata in data använder man nyckelordet `indata`. Detta nyckelord används som en variabel och gör att när exekveringen av programmet når den punkt där nyckelordet finns kommer programmet att stanna upp och vänta på att användaren matar in något via tangentbordet, följt av en radbrytning.

Vill man till exempel låta användaren mata in sitt namn kan det se ut såhär:

```
skriv "Vad heter du?"
namn blir indata
```

Detta kommer göra att programmet skriver ut "Vad heter du?" (utan citattecken), väntar på att användaren ska mata in något, och sedan sparar användarens inmatning i variabeln namn.

Programflöde och villkorssatser

Att kontrollera programflödet och ge villkor för vad som ska hända är grundläggande inom programmering. För att ge ett villkor till ett kodavsnitt använder Seraphina nyckelordet `om`. Nyckelordet används genom att skriva `om` *`villkor`* `kör`, där *villkor* är en jämförelse mellan två data. Det följande ordet `kör` är ännu ett nyckelord som används tillsammans med `slut` för att kapsla in kod i så kallade block. All kod som är inom samma block kommer att påverkas av satsen innan, och i det här fallet endast köras om villkoret är sant.

Den förmodligen vanligaste jämförelsen, `är`, används för att se om två givna data har samma värde. Det går till exempel att undersöka om värdet på `a` är `1`:

```
a blir 2
om a är 1 kör
skriv "a är ett"
slut
```

I det här fallet var `a` inte `1`, så utskriften kommer inte att genomföras.

I Seraphina finns det totalt fyra olika jämförelseoperatorer:

```
är
är mindre än
är större än
inte är.
```

För att jämföra `heltal` eller `decimaltal` fungerar alla fyra av dessa, men för att jämföra `text` fungerar endast `är` och `inte är`.

För att ha flera jämförelser i samma villkor,för att se om `a` är större än eller lika med `b`, används så kallade logiska operatorer. I Seraphina stöds operatorerna `och`, `eller` och `inte`, och används på följande sätt:

```
a blir 2
b blir 3
om a är större än b eller a är b kör
     skriv "a är större än eller lika med b"
slut
om a är mindre än b och a är större än 0 kör
     skriv "a är mindre än b, men större än noll"
slut
om a är mindre än b och a är mindre än 0 kör
     skriv "a är mindre än både b och noll!
slut
```

Den första utskriften kommer i detta fall inte ske, eftersom `a` är mindre än `b`.

Det finns även ett annat sätt att göra den undre utskriften i exemplet ovan. Eftersom a inte är större än b, och a inte är lika med b, så måste a vara mindre än b. Ett sätt att skriva detta är genom att använda nyckelordet `annars`, vilket skulle göra att ovanstående exempel istället skrivs så här:

```
a blir 2
b blir 3
om a är större än b eller a är b kör
     skriv "a är större än eller lika med b"
slut
annars om a är större än 0 kör
     skriv "a är mindre än b"
slut
annars kör
     skriv "a är mindre än både b och noll!"
slut
```

Vi rekommenderar att detta sätt används om möjligt, eftersom det gör att det inte blir lika mycket kod att läsa eller skriva, och därför också lättare att förstå.

Om man skulle vilja skriva ut ett stycke text flera gånger på skärmen kan man naturligtvis göra det genom att skriva:

```
skriv “Text nummer 1”
skriv nyrad
skriv “Text nummer 2”
skriv nyrad
skriv “Text nummer 3”
skriv nyrad
```

Detta sätt att skriva på blir dock snabbt jobbigt, så det finns ett smidigare sätt att göra det på. Med nyckelordet `medan` skapar man ett kodavsnitt som kommer att köras så länge villkoret stämmer. Om man använder det för att skriva om ovanstående exempel ser koden istället ut så här:

```
a blir 1
medan a är mindre än 4 kör
     skriv “Text nummer “
     skriv a
     skriv nyrad
     a blir a + 1
slut
```

Den här metoden gör att det blir mer kod till att börja med, men om antalet utskrifter ökas till 10, 100 eller 1000 blir koden inte längre, till skillnad från det första exemplet där varje extra utskrift gör koden två rader längre. Detta gör att det senare sättet är att föredra.

Funktioner

En annan viktig konstruktion inom många program är funktioner. En funktion är ett avsnitt kod som kan anropas och köras flera gånger. Funktioner kan även ta in värden som parametrar och använda sig av dessa. Funktioner kan också returnera värden till den del av koden som anropade funktionen. Funktioner i Seraphina skapas på följande sätt:

```
funktion funktionsnamn_ett kör
     skriv “Text som skrivs när funktionen anropas.”
slut
funktion funktionsnamn_två använder a och b kör
     skicka tillbaka a + b
slut
```

Först skrivs alltså `funktion` för att visa att det som följer är en funktion. Sedan skrivs ett namn som används för att anropa funktionen. Efter namnet finns möjligheten att säga att funktionen ska ta in värden att använda sig av. Detta görs genom att man skriver `använder` följt av variabelnamn, vilka man använder för att komma åt datan som skickas in i funktionen. Om fler än en variabel ska användas separeras dessa med `och`. Slutligen skrivs själva koden som funktionen ska utföra, inkapslad i ett block av `kör` och `slut`. Vill man att funktionen ska returnera data till den plats där den anropades skriver man `skicka tillbaka` följt av det som ska skickas tillbaka. I exemplet är detta ett uttryck som blir resultatet av en addition av de två värdena funktionen får in.

För att sedan anropa en funktion och använda den någonstans i koden skrivs helt enkelt namnet på funktionen. Vill man anropa en funktion som använder parametrar säger man att funktionen ska anropas `med` den data man vill skicka in. Koden som följer kommer alltså först skriva ut "Text som skrivs när funktionen anropas." och sedan siffran 8 (med de funktionsdefinitioner som angavs i förra exemplet):

```
funktionsnamn_ett
skriv funktionsnamn_två med 3 och 5
```

En sak som är viktig att tänka på då man jobbar med funktioner är att alla variabler som deklareras och används i en funktion är unika till just den funktionen och kan inte användas utanför den. Det går med andra ord att skriva såhär:

```
funktion skriv_ett kör
     x blir 1
     skriv x
slut
x blir 5
skriv_ett
```

Detta ger utskriften "1", då funktionen kommer använda sig av den variabel x som ligger "närmast" i räckvidd, det vill säga i samma funktion. Detta gör också att det inte är några problem att skapa en funktion med namn på parametrar som redan används utanför funktionen så länge man är beredd på att man inte kommer kunna komma åt de "yttre" variablernas värden. Finns inte variabeln angiven i funktionen kommer värdet tas från det närmaste steget uppåt. Med andra ord, i följande exempel skulle utskriften bli 5:

```
funktion skriv_x kör
     skriv x
slut
x blir 5
skriv_x
```

Funktionen kommer köras och vilja skriva ut värdet av variabeln `x`. Då denna variabel inte finns i funktionen används värdet av `x` utanför funktionen. Vi kan i funktioner komma åt variabler utanför dem så länge vi inte har egna med samma namn i dem, men det går aldrig att ändra en yttre variabels värde i en funktion. Detta skulle istället skapa en ny variabel, unik för funktionen, med samma namn och det nya värdet.

Systemdokumentation

Seraphina är implementerat i Ruby och använder sig till stor del av den givna parsern `rdparse.rb`, vilken är skriven av Peter Dalenius, adjunkt vid Institutionen för datavetenskap på Tekniska högskolan vid Linköpings universitet. Eftersom den är skriven av en tredjepart kommer dess uppbyggnad och funktion inte diskuteras närmare i denna rapport.

Grammatik

Seraphinas grammatik är egentligen väldigt lik en annan grammatik som vi hade studerat innan vi skrev vår egen, nämligen grammatiken för Python. Python är dock ett mycket mer komplicerat språk än Seraphina, och har därför också mycket mer komplicerad grammatik. Grammatiken för Seraphina blir generellt mer grundläggande då språket inte behöver ha lika många möjligheter för olika konstruktioner som Python. Grammatiken finns i sin helhet i Bilaga 1.

Programexekvering

När ett program i Seraphina ska exekveras sker detta i ett antal steg. Nedan kommer dessa steg tas upp indelat efter de filer som sköter respektive steg. All kod finns bifogad dessa dokument, se bilaga 2-4.

"seraphina.rb"

För att exekvera ett program körs först programfilen `seraphina.rb`. Detta kan göras med eller utan argument och programmet har två olika beteenden beroende på hur det startas. Om programmet startas med ett filnamn som argument kommer `seraphina.rb` se till att filen som angetts parsas och sedan exekveras. Körs `seraphina.rb` utan argument startas istället den interaktiva tolken som ger användaren en prompt och rad för rad parsar och exekverar det användaren matar in.

"rules.rb"

När `seraphina.rb` ska parsa koden som matas in skapas en instans av en klass som definieras i filen `rdparse.rb`. Denna fil är i sin tur inkluderad i filen `rules.rb` som innehåller de delar av parsningskoden som vi skrivit. Som nämnts tidigare är själva parsern skriven av en tredje part och därför kommer vi inte gå in närmare på exakt hur den fungerar. Filen `rules.rb` innehåller dock dels, som namnet antyder, en uppsättning regler för parsern, dels koden för exekvering.

Parsningen går till så att parsern från `rdparse.rb` först tokeniserar hela den inmatade koden utifrån regler angivna med reguljära uttryck på raderna 13 till och med 28 i `rules.rb`. När tokeniseringen är klar går parsern genom de tokens den skapat, i ordning, och matchar mönster mot reglerna mellan rad 31 och 226 i samma fil. Dessa regler talar om vilka mönster av tokens som ska matchas mot vad och även vad som ska göras när ett specifikt mönster påträffas. Denna regeluppsättning är i princip Seraphinas BNF-grammatik skriven i kodform, utökat med kod för att hantera det som påträffas.

För att kunna exekvera koden efter parsningen låter vi här, med hjälp av dessa regler, parsern bygga upp ett träd av noder, olika för olika typer av uttryck och statements. Till exempel, om parsern stöter på ett mönster av tokens som matchar en tilldelning skapas en nod av typen `AssignmentNode` som innehåller dels namnet på variabeln som tilldelas, dels värdet eller uttrycket som ska tilldelas till variabeln. Är det som ska tilldelas i sin tur ett uttryck kommer parsern ha matchat detta tidigare och en `ExpressionNode` innehållandes uttrycket kommer bindas till noden som skapas. Slutligen placeras alla grundnoder (det vill säga noder som inte binds till andra noder över sig) i en lista.

När sedan koden ska exekveras går funktionen `run()` genom listan som skapats och anropar funktionen `eval()` för varje nod.

“nodes.rb”

Noderna som skapas av parsern definieras i filen `nodes.rb`. Det är i noderna koden för att faktiskt exekvera Seraphina-kod finns. Alla noder skapas med argument för den data som har med dem att göra, som nämnts ovan skapas till exempel en `AssignmentNode` med ett namn och ett värde eller en bindning till den nod som ska tilldelas.

När sedan exekveringen sker kallar funktionen `run()` på varje nods `eval()`-funktion. Denna funktion är i sin tur den som utför det koden ska göra. För att återgå till exemplet med en `AssignmentNode` kommer denna funktion först att skapa en plats för den nya variabeln i rätt scope med det angivna namnet. Därefter kommer funktionen `eval()` att anropas för alla eventuella subnoder, i fallet med tilldelning enbart värdet. Att kalla på funktionen eval() är viktigt eftersom det som noden skapats med i sin tur skulle kunna vara ett uttryck eller liknande istället för ett rent värde och då måste evalueras innan tilldelningen sker. Vid exekvering körs noderna i grundlistan var för sig och för varje nod anropas alla eventuella subnoder nedåt i trädstrukturen. När de noder som ligger djupast i trädstrukturen evalueras kommer de att börja skicka tillbaka värden, och nodträden kommer exekveras färdigt nerifrån och upp.

“stl.sp”

I filen `stl.sp` finns Seraphinas standardbibliotek. Standardbiblioteket är utrymme för att implementera funktioner som kan tänkas användas i många olika program. Funktioner i standardbiblioteket har man alltid tillgång till, oavsett var man är i koden. Funktionerna som är definierade i standardbiblioteket går att överlagra om man vill, men för att minimera risken för förvirring bör detta undvikas.