SproutScript

Version 1.0
TDP019: Projekt: Datorspråk
Linköpings universitet

Love Arreborn - lovar063@student.liu.se
Elliot Hernesten - ellhe126@student.liu.se

23 november 2023

Innehåll

1 Introduktion

Detta dokument är en genomgående systemdokumentation för programmeringsspråket SproutScript, som skapats av Love Arreborn och Elliot Hernesten. Språket konstruerades för kursen TDP019: Projekt: Datorspråk[1], en kurs för programmet Innovativ Programmering på Linköpings universitet. Projektet pågick under vårterminen 2023.

1.1 Projektets förutsättningar

Projektets omfattning beskrivs i kursplanen. Kort sammanfattat så förväntas studenterna att efter detta projekt att kunna:

- konstruera ett mindre datorspråk
- diskutera och motivera designval i det egna datorspråket med utgångspunkt i teori och egna erfarenheter
- implementera verktyg (interpretator, kompilator, etc) för det egna datorspråket
- formulera teknisk dokumentation av det egna datorspråket
- presentera en uppgift muntligt inför publik enligt angivna förutsättningar

Källa: TDP019: Projekt: Datorspråk, IDA, Linköpings universitet[2]

Arbetet med detta projekt pågick mellan februari - maj 2023.

1.2 Projektbeskrivning

Kursens mål var att producera ett enklare programmeringsspråk. Detaljer kring språkets omfattning, syntax och funktionalitet var fria, vissa krav för specifika betygsnivåer specificerades på kurshemsidan[2].

Idén med SproutScript var att skapa ett enklare programmeringsspråk riktat till nya programmerare. Ambitionen var att inkludera enklare aspekter från exempelvis Python eller Ruby, samt att inkludera viss striktare syntax med inspiration från C++ eller C.

[1] https://studieinfo.liu.se/kurs/TDP019
[2] https://www.ida.liu.se/~TDP019/current/projekt/index.sv.shtml

2 SproutScript - produktbeskrivning

SproutScript är ett simpelt programmeringsspråk byggt på Ruby. Språket är designat för att ge en enkel instegssport för nya programmerare att lära sig enklare koncept, samt att förstärka goda syntaktiska vanor som är fördelaktiga att lära sig som nybörjare.

Det är ett löst typat språk som inte kräver strikt kännedom kring datatyper för exempelvis variabeltilldelning eller returvärden, men förstärks av striktare syntax-regler såsom blockavgränsning med klammerparenteser samt radavslut med semikolon.

SproutScript är inte menat att vara ett fullskaligt programmeringsspråk som ska kunna nyttjas för att skriva stora, komplexa program i - det är snarare ett språk som ämnar att vara ett bra första steg in i programmering för en nybörjare. SproutScript har tydliga begränsningar i sin omfattning, och när en användare når dessa begränsningar är det en god idé att växla sitt fokus till ett annat språk såsom Python.

2.1 Installation och snabbstartsguide

SproutScript är byggt på Ruby version 3.1.4p223, och har inte testats för äldre versioner. Information om hur Ruby kan installeras för alla typer av maskinvara hittas med fördel på Rubys dokumentationssida[3]. Att ha Ruby installerat är ett krav för att kunna använda SproutScript. SproutScript kan sedan laddas ned packeterat i en ZIP-fil genom denna länk[4]. Extrahera ZIP-filen i valfri katalog på din dator.

2.2 Köra SproutScript-filer

SproutScript kan endast köra filer med filändelsen .sps och förutsätter att dessa filer är skrivna enligt den syntax som beskrivs i detta dokument.
Användandet av SproutScript förutsätter viss grundläggande kunskap kring hur en användare kan navigera mellan kataloger genom sitt operativssystems terminal. Du som är ovan vid detta kan hitta grundläggande instruktioner för

[3] https://www.ruby-lang.org/en/documentation/installation/

[4] https://liuonline-my.sharepoint.com/:u:/g/personal/lovar063_student_liu_se/EROjoysDD9JOsSr3wLLnUA8Bre7x5wZT_gjsr-JIFOeGeQ?e=e0u5Ua

Windows[5], Linux[6] och MacOS[7].

2.2.1 Windows

Vi rekommenderar att Ruby installeras genom RubyInstaller[8], och samtliga kommandon nedan förutsätter en installation genom denna metod.

I Windows Command Prompt, navigera till den extraherade SproutScript-katalogen på din dator. Med fördel placeras egenskrivna .sps-filer i under-mappen projects". En .sps-fil kan därefter köras med följande kommando:

```
C:\Ruby\bin\ruby.exe sproutscript.rb projects/[din fil].sps
```

2.2.2 Linux och MacOS

Hur du installerar Ruby kan variera från distribution till distribution om du arbetar i Linux. Kontrollera den version av Ruby som du har installerad innan du fortsätter. Vi rekommenderar att göra SproutScripts huvudfil till en körbar fil, så att du enklare kan köra dina filer.

```
ruby -v
ruby 3.1.4p223 (2023-03-30 revision 957bb7cb81) [x86_64-linux]
chmod +x sproutscript.rb
./sproutscript.rb [din fil].sps
```

2.2.3 Kommandoradsargument

SproutScript inkluderar ett par hjälpsamma kommandoradsargument som man kan lägga till innan sin fil, för att ge lite extra funktionalitet.

```
./sproutscript.rb --help // Visar alla möjliga argument
./sproutscript.rb --log [din fil].sps // Visar en debug-logg
./sproutscript.rb -- tree [din fil].sps // Visar hela syntax-trädet som ska k
```

[5] https://www.lifewire.com/change-directories-in-command-prompt-5185508
[6] https://www.redhat.com/sysadmin/navigating-filesystem-linux-terminal
[7] https://appletoolbox.com/navigate-folders-using-the-mac-terminal/
[8] https://rubyinstaller.org/

SproutScript - Användarhandledning

Denna sektion är skiven för en ny programmerare, och kommer att beskriva de koncept som SproutScript har stöd för i nära detalj. För dig som är van vid programmering, och vill ha en snabb introduktion till hur du skriver SproutScript-kod, vänligen se sektion 8.

När du ska skriva SproutScript-kod kommer du att göra det i en fil som har suffixet `.sps`. En viktig detalj är att alla rader som skrivs i dokumentet måste avslutas med ett semikolon.

```
print("Hej!");
```

3 Datatyper

Konsten att skriva programkod handlar om att bearbeta och manipulera data. Data i SproutScript tillhör alltid en viss datatyp, som kan användas för ett flertal olika operationer. Denna sektion beskriver vilka datatyper som existerar samt vilka operationer som kan nyttjas på dessa datatyper (sektion 3.2, sektion 3.3 samt sektion 3.4). Hur en användare vidare kan bearbeta och manipulera data kommer därefter att beskrivas i sektion sektion 4.

3.1 Introduktion till datatyper

Data kan klassificeras i två generella typer – enkla datatyper och komplexa datatyper. I mer avancerade programmeringsspråk har dessa datatyper större betydelse än vad de har i SproutScript, då en enkel datatyp i regel tar upp mindre minnesplats på en dators interna minne när ett program körs, medan en komplex datatyp ofta är en kombination av enkla datatyper i en specifik struktur. Uppdelningen av datatyper i SproutScript är i praktiken en formalitet som presenteras i denna form för att förbereda användaren för andra programmeringsspråk, där typen av data kan ha mer betydelse.

Notera att samtliga datatyper nedan kommer att beskrivas med sina engelska namn i första hand för att bygga upp en terminologi hos användaren som går att överföra till andra programmeringsspråk.

3.2 Variabeltilldelning

Enkla datatyper definieras av att de består av en liten mängd information, vilket låter dem ta en jämförelsevis liten del av en dators interna minne. När

en programmerare skriver kod som använder en datatyp kommer denna data att lagras i datorns interna minne (RAM-minne) på en specifik minnesadress. Man kan se en variabel som en låda, och i den lådan kan man lägga ett speciellt värde. Sedan kan man titta i den lådan för att få tag på detta värde.

För att en programmerare enkelt ska kunna arbeta med dessa datatyper efter att de har skapats så kan enklare datatyper tilldelas ett variabelnamn. När man i kod skriver ut en variabel – lådan som man skapar – och tilldelar denna variabel ett värde - säger vad som ska ligga i lådan - kallas det för att deklarera en variabel. Detta uttryck kommer att nyttjas framöver.

I SproutScript har variabelnamn vissa strikta regler:

- Namnet måste börja med en liten bokstav
- Namnet kan endast innehålla bokstäver (både gemener och versaler), siffror och bindestreck.

En datatyp kan tilldelas ett valfritt variabelnamn som inte ingår i listan med skyddade ord. Alla datatyper kan tilldelas ett variabelnamn för att ge enkel åtkomst till dem senare, se listing 1.

```
variable = 10;    // Funktionen print skriver ut
print(variable);  // värdet i terminalen
==> 10 // Detta motsvarar utskriften som sker i terminalen
```

Listing 1: Variabeln deklareras med värdet 10.

Varje variabelnamn kan endast innehålla ett värde i taget. Undantaget för detta är när man arbetar i ett separat scope, vilket täcks i sektion 5.3. Det är möjligt att uppdatera värdet på en variabel genom att göra en ny tilldelning, se listing 2.

```
variable = 10;
print(variable);
==> 10
variable = 20;
print(variable);
==> 20
```

Listing 2: Värdet 10 skrivs över av värdet 20 i den andra tilldelningen.

God praxis är att ge variablerna namn som tydligt beskriver vad de förväntas innehålla, och att undvika variabelnamn som är enbart en bokstav. Om en variabel behöver ett längre namn rekommenderas det att inkludera understreck i namnet för att öka läsbarheten, se listing 3.

```
current_value = 10;
print(current_value);
==> 10
```

Listing 3: En variabel tilldelas med annat, längre namn.

3.3 Enkla datatyper

Inom programmering delas olika typer av data generellt upp i två typer – enklare datatyper och komplexa datatyper. Som noterat i sektion 3.1 så är denna uppdelning inte av lika stor vikt i SproutScript som i andra programmeringsspråk, men det är nyttigt att bekanta sig med de olika datatyperna som finns och hur de interagerar med varandra.

3.3.1 Integers

Denna datatyp handlar om heltal. Variabler av denna typ kan innehålla vilket reellt heltal som helst. Man kan tilldela ett heltal till en variabel ensamt, eller så kan man tilldela en variabel värdet av en uträkning.

```
integer = 10 * 10;
print(integer);
==> 100
```

Listing 4: En variabel tilldelas med produkten av 10 * 10.

För matematiska tal kan man arbeta med alla vanliga aritmetiska operatorer, likt i listing 4. En fullständig tabell över alla matematiska operatorer som kan användas i SproutScript samt kodexempel på hur dessa används återfinns i sektion 4.

3.3.2 Floats

Floats, eller flyttal, syftar till decimaltal – när vi inte är intresserade av att arbeta med exklusivt heltal. Det är ofta fördelaktigt att arbeta med heltal, då

detta i regel räcker utmärkt för de operationer som generellt behöver göras, men det finns tillfällen då mer precision krävs. För dessa situationer kan man arbeta med flyttal. Att arbeta med pi, som i lising 6 och 7, är ett konkret exempel på detta.

```
pi = 3.14159265359;
print(pi);
==> 3.14159265359
```

Listing 5: Variabeln pi deklareras.

Likt heltal så fungerar alla aritmetiska operatorer även för flyttal. En fullständig tabell över dessa återfinns i sektion 4. När man utför en matematisk operation med ett heltal och ett flyttal så kommer resultatet av denna att vara ett nytt flyttal.

```
pi = 3.14159265359;
radius = 10;
area = pi * radius ^ 2;
print(area);
==> 314.159265359
```

Listing 6: Arean på en cirkel med radien 10 kalkyleras.

3.3.3 Booleans

I programmering är det vanligt att arbeta med sanningsvärden. Något som vi kommer att stöta på i sektion sektion 5 är ett sätt att kontrollera vilken kod som exekveras, eller körs, beroende på ett visst villkor. Som ett konkret exempel så kan man jämföra två stycken heltal, och fråga sig om ett tal är större än det andra. Om så är fallet får man ett booleskt sanningsvärde – och om så inte är fallet får man istället ett falskt booleskt värde.

Normalt får man dessa värden genom en jämförelseoperation eller dylikt, något som täcks i sektion 4, men man kan även explicit deklarera ett värde som sant eller falskt i ren kod.
Värt att notera är att andra datatyper också kan utvärderas som booleska värden. Alla datatyper klassas som ett booleskt sant värde - med ett undantag. Siffran 0 kommer alltid att utvärderas som falskt. Detta är en intressant egenskap som kan komma att vara hjälpsam under sektion 5, när vi tittar på styrstrukturer och loopar.

```
programming_is_fun = True;
programming_is_hard = False;
```

Listing 7: Variabler tilldelas med explicit sanna och falska värden.

3.4 Komplexa datatyper

De enkla datatyper som vi har tittat på hittills är enkla datatyper, som i praktiken består av små värden i en dators interna minne. Komplexa datatyper är däremot en sammansättning av flera stycken enkla datatyper i en speciell struktur.

3.4.1 String

Konceptet med strängar refererar i praktiken till ett eller flera ord som behöver skrivas ut eller manipuleras i ett datorprogram. Man kan se dessa som en lista med individuella karaktärer som sätts ihop till ett stycke data. Varje individuell karaktär kan alltså ses som en bit enkel data, och tillsammans skapar de en komplex datatyp. Komplexa datatyper, precis som enkla datatyper, kan enkelt tilldelas till en variabel. När man deklarerar en sträng behöver man omge strängen med citattecken, såsom exemplet i listing 8. Allting inom dessa citattecken blir innehållet i strängen.

```
introduction = "Hello world!";
print(introduction);

==> Hello world!
```

Listing 8: Strängen `Hello world!` skrivs ut i terminalen.

Det är även möjligt att använda vissa operatorer som definieras i sektion 4 med strängar. Till exempel kan man enkelt lägga ihop två strängar, samt multiplicera en sträng flera gånger innan den läggs till.

I listing 9 ser vi att tre olika strängar deklareras. Därefter används den aritmetiska additionsoperatorn för att lägga ihop flera strängar till en. Detta kallas för att sammanfoga en sträng, eller *concatenate* på engelska. När två strängar sammanfogas bör man som programmerare vara uppmärksam på hur de kommer att sättas ihop, och därför läggs det till ett mellanslag i strängen som ska sammanfogas. Hur resultatet blir utan dessa mellanslag framgår i listing 10.

```
part_one = "Make me "; // Observera mellanslaget
part_two = "really ";  // i slutet av strängen!
part_three = "large!";

full_sentence = part_one + part_two * 4 + part_three;

print(full_sentence);

==> Make me really really really really large!
```

Listing 9: Flera strängar kombineras och upprepas med aritmetiska operationer.

```
my_string = "Make Me" + "really" * 4 + "large!"
print(my_string);

==> Make Mereallyreallyreallyreallylarge!
```

Listing 10: Flera strängar kombineras, men utan mellanslag.

3.4.2 Lists

Inom programmering är det vanligt att vilja ha ett set av data i en strukturerad ordning, något som i SproutScript görs bäst med hjälp av listor. En lista är en typ av databehållare, som i sin tur kan innehålla en godtycklig mängd data. Varje bit data i listan kallas generellt för ett *element*. En lista kan deklareras genom att placera ut matchande hakparenteser. När listan deklareras kan man även inkludera en godtycklig mängd element som den ska innehålla – och hur detta ska se ut i praktiken framgår i listing 11.

```
empty_list = [];
variable = 10;
list_with_elements = [1, 1.2, "Hello", variable];
```

Listing 11: En tom lista deklareras, samt en lista med element.

Man kan komma åt ett specifikt element i listan genom att ange det elementets index-värde inom hakparenteser direkt efter listans variabelnamn (eller listan själv), se exempel på detta i lising 12. Inom programmering så börjar alla typer av listor på 0, snarare än ett.

```
index_example = [1, 2, 3, 4];
// Index 0 blir första elementet i listan, alltså 1
print(index_example[0]);
==> 1

print(index_example[1);
// Index 1 blir det andra elementet i listan, alltå 2
==> 2
```

Listing 12: En tom lista deklareras, samt en lista med element.

Till listor finns det funktioner som man kan använda för att bland annat lägga till och ta bort element ur en lista, samt rensa alla element vid behov. Exakt vad en funktion kommer att beskrivas i mer detalj i sektion 6 – i denna sektion kommer vi endast att beskriva de funktioner som är unika för listor.

För att använda en funktion så skriver man dess namn, följt av parenteser. I parenteserna placerar man ett korrekt antal parametrar - data som ska användas av funktionen. Därefter kommer funktionen att utföra sin operation, och skicka tillbaka ett returvärde. Betrakta hur de list-specifika funktionerna används i listing 13.

```
my_list = [];
append(my_list, 1); // lägger till siffran 1 i listan
append(my_list, 2); // lägger till siffran 2 i listan
print(my_list);
==> [1, 2]

popped = pop(my_list); // tar bort sista elementet i listan
// returvärdet sparas till variabelnamnet popped
print(my_list);
==> [1]

print(popped);
==> 2
```

Listing 13: Två listmetoder demonstreras.

I tabell 1 och 2 beskrivs alla funktioner som fungerar för listor i SproutScript. I tabell 1 beskrivs append, pop, sort och i tabell 2 beskrivs clear, delete at och index. I tabellerna beskrivs vad funktionerna gör, vad de returnerar och exempel på hur de kallas på i språket.

Tabell 1: Listfunktionerna `append`, `pop` och `sort`

Append	**Pop**	**Sort**
Lägger till angivet element i listan	Tar bort sista element i listan	Sorterar listan efter storleksordning
Returnerar en lista med det nya värdet	Returnerar det raderade elementet	Returnerar den sorterade listan
Används för att lägga till element i listan	Används för att ta bort det sista elementet i listan	Används för att kunna sortera en lista efter t.ex storlek
Ex. append([1, "Hello", 7, 8, 6, 5], True)	*Ex. pop([1, "Hello", 7, 8, 6, 5, True])*	*Ex. sort([1, 2, 3])*
Resultat: [1, "Hello", 7, 8, 6, 5, True]	**Resultat: [1, "Hello", 7, 8, 6, 5]**	**Resultat: [3, 2, 1]**

Tabell 2: Listfunktionerna `clear`, `delete_at` och index-åtkomst

Clear	**Delete at**	**Index**
Rensar listan	Tar bort ett element efter angivet index	Hämtar ett element i listan efter angivet index
Returnerar en tom lista	Returnerar det raderade elementen vid index	Returnerar värdet vid index
Används för att enkelt och effektivt kunna tömma en lista	Används för att enkelt kunna ta bort ett specifikt element ur listan	Används för att enkelt kunna hämta värden ur listor
Ex. clear([1, "Hello", 7, 8, 6, 5])	*Ex. delete_at([1, "Hello", 7, 8, 6, 5], 1)*	*Ex. a = [3,2, "Hello"] a[2]*
Resultat: []	**Resultat: [1, 7, 8, 6, 5]**	**Resultat: Hello**

4 Operatorer och jämförelser

För att de datatyper som beskrivits i sektion 3 ska vara användbara så krävs det möjlighet att manipulera och utvärdera datan. Ett fåtal aritmetiska operatorer har redan visats i föregående sektion, och i denna kommer vi att beskriva samtliga enkla operatorer som går att använda på de enkla datatyperna.

4.1 Aritmetiska operatorer

Aritmetiska operatorer används för att ändra på värdet av en datatyp. Generellt nyttjas dessa för matematiska operationer med reella tal, men som det visats i sektion 3.4.1 så kan ett fåtal aritmetiska operatorer även användas för att

manipulera strängar. Tabell 3 visar samtliga aritmetiska operatorer, samt vilken matematisk prioritet de har i SproutScript. Endast + och ∗ går att använda på strängar.

Tabell 3: Operatorer och prioritering

Operatorer	Prioritering
+	4
-	4
*	3
/	3
%	3
^	2
()	1

4.2 Jämförelseoperatorer

När data manipuleras är det intressant att utvärdera det nya resultatet för att besluta nästa steg i programmet. Genom att jämföra två instanser av data kan vi utföra olika operationer beroende på värdet på en viss variabel, något som beskrivs i mer detalj i sektion 5.

Dessa jämförelser efterliknar de som finns inom matematiken, och skrivs enligt de exempel som listas i tabell 4. Varje jämförelse resulterar i ett booleskt värde (se sektion 3.3.3) - en jämförelse resulterar antingen i sant eller falskt.

Tabell 4: Jämförelseoperatorer

Operatorer	Betydelse
==	Lika med
!=	Inte lika med
>	Större än
>=	Större eller lika med
<	Mindre än
<=	Mindre eller lika med

4.2.1 Exempel på jämförelseoperatorer

Sektion 5 kommer att visa på konkreta exempel på hur dessa jämförelseoperatorer kan användas. I listing 14 visar vi dock hur de väldigt enkelt kan användas – detta är dock ett ytterst ovanligt sätt att använda en jämförelse på.

```
first = 10;
second = 20;

print(first != second);

==> True

print(first > second);

==> false
```

Listing 14: Exempel på enklare jämförelser.

4.3 Logiska operatorer

I vissa fall räcker det med att göra en enkel jämförelse för att kunna styra programmet vidare. Andra gånger så behöver två värden jämföras samtidigt, och nästkommande operation beror på dessa två jämförelser samtidigt. I dessa fall kan man nyttja en logisk operator för att utvärdera två jämförelser på samma gång, och få ett booleskt värde som representerar den kombinerade jämförelsen.

4.3.1 Exempel på logiska operatorer

Logiska operetorer kan skrivas både textat och med deras respektive symboler. Detta innebär att man kan välja att använda ett eller bägge alternativen när man programmerar i SproutScript. Det rekommenderas dock att använda en av metoderna för att vara konsekvent. I tabell 5 framgår samtliga sätt som en logisk jämförelse kan skrivas i SproutScript, vilket även visades praktiskt i listing 15.

```
first = 10;
second = 20;
print(first != second or first > second);

==> True

print(first != second and first > second);

==> False
```

Listing 15: Två stycken jämförelser med logiska operatorer.

Tabell 5: Logiska operatorer

Jämförelseoperatorer	Textat	Betydelse
&&	and	'och' Båda sidorna måste utvärderas till sanna.
||	or	'Eller' en av sidorna måste utvärderas till sanna.
!	not	Inte värdet, motsatts.

5 Styrstrukturer och loopar

Genom att manipulera och utvärdera data så kan programmeraren enkelt bygga upp en struktur som exekverar specifika delar kod beroende på resultatet av en jämförelse, eller upprepa en operation tills dess att ett villkor har uppfyllts. Dessa verktyg i programmerarens arsenal är enormt kraftfulla, och det är tack vare möjligheten att ställa upp styrstrukturer och loopar som möjliggjör konstruktion av även de mest komplexa programmen – trots att de fundamentala byggstenarna är tämligen simpla.

5.1 Styrstrukturer

Med styrstrukturer kan programmeraren skriva block av kod som endast kommer att köras om en viss jämförelse är sann. Genom att placera ut en eller flera sekvenser av styrstrukturer är det möjligt att bearbeta och manipulera data, och beroende på utfallet kommer olika block av kod att köras. Se listing 16 för ett grundläggande exempel.

```
if (x > 10) {
    print("Hello");
}
```

Listing 16: En enkel styrstruktur.

Detta är den enklaste varianten av en styrstruktur. All kod som ligger inom klammerparenteserna kommer endast att exekveras om villkoret inom parenteserna är sant. Skulle variabeln `x` ha ett värde lägre än 10 kommer utskriften alltså inte att ske. Det finns även möjlighet att placera fler styrstrukturer i en kedja.

```
if (x > 10) {
    print("Hello");
} else if (x == 10) {
    print("Goodbye");
} else {
    print("See you!");
}
```

Listing 17: En större styrstruktur med fler villkor.

I listing 17 är den tidigare styrstrukturen från listing 16 förlängd, och vi har fler villkor att betrakta. Denna struktur – `if`, `else if`, `else` – är de enda styrstrukturerna som finns i SproutScript, och de är även de vanligaste att arbeta med i andra programmeringsspråk. Dessa villkor kommer att utvärderas i ordning, och det första villkoret som är sant är det block av kod som kommer att köras. Om inget villkor i styrstrukturen är sant, så kommer det blocket kod som står under grenen `else` att köras. Betraktar vi koden ovan, så ser vi alltså att följande kommer att ske:

1. Har variabeln `x` ett värde strikt större än 10, så kommer `"Hello"` att skrivas ut i terminalen.
2. Har variabeln `x` det exakta värdet 10, så kommer `"Goodbye"` att skrivas ut i terminalen.
3. Om värdet på talet `x` inte är större än 10, och inte är exakt 10, så kommer `"See you!"` att skrivas ut i terminalen.

5.1.1 If-satser

En styrstruktur måste alltid inledas med en `if`-sats. Denna `if`-sats måste i sin tur alltid följas av någon form av jämförelse eller logisk operator som ger

ett sant eller falskt värde. Eftersom en siffra kan utvärderas till antingen sant eller falskt beroende på dess värde, så kan man även utvärdera detta – se exemplet i listing 18.

```
x = 1;

if (x) {
    print("Hello");
}

y = 0;

if (y) {
    print("Goodbye");
}

==> Hello
```

Listing 18: En ensam variabel skickas in i styrstrukturen.

Eftersom variabeln `x` har ett värde – 1 – så klassas den som ett booleskt sant värde. Variabeln `y` deklareras å andra sidan med värdet 0, vilket ger ett booleskt falskt värde. Detta innebär att endast `"Hello"` skulle skrivas ut från koden i listing 20.

Det är även möjligt att skriva en **`if`**-sats inuti en annan **`if`**-sats – något som kallas för nästling. Betrakta det följande kodexemplet i listing 19:

```
if (x > 10) {
    if (x > 15) {
        if (x > 20) {
            print("Hello");
        }
    }
}
```

Listing 19: En nästlad styrstruktur.

Variabeln `x` måste i detta exempel ha ett värde strikt större än 20 för att samtliga nästlade jämförelser ska vara sanna, och utskriften ska ske.

5.1.2 Else if-satser

En `if`-sats kan följas av en `else if`-sats om användaren vill göra en ytterligare jämförelse. Detta ger programmeraren möjlighet att skriva en kedja av utvärderingar, och vara säker på att endast ett av alternativen – en gren av styrstrukturen – kommer att köras. Det är i praktiken möjligt att skriva ett obegränsat antal `else if`-satser i en kedja.

```
x = 45;

if (x > 10) {
    print("Start");
} else if (x > 20) {
    print("Branch 1");
} else if (x > 30) {
    print("Branch 2");
} else if (x > 40) {
    print("Branch 3");
} else if (x > 50) {
    print("Branch 4");
}

==> Branch 3
```

Listing 20: En längre styrstruktur.

I listing 20 kan vi se att endast en gren i styrstrukturen körs, trots att de två sista jämförelserna också är sanna. Detta beror på att endast ett alternativ i en styrstruktur kan exekveras. Så fort ett av villkoren är sant så körs den koden som ligger i det blocket, och så lämnar programmet styrstrukturen. Skulle variabeln `x` ha värdet 5 i listing 20 så skulle ingenting skrivas ut i terminalen, då ingen del av styrstrukturen matchar.

5.1.3 Else-satser

Om programmeraren skriver en styrstruktur med flera jämförelser så finns det möjlighet att inget av de villkoren som definierats är sanna. I dessa tillfällen så kan man avsluta sin styrstruktur med en `else`-sats. Observera att det som mest kan finnas en `else`-sats per `if`-sats.

Som vi noterade i listing 20 i föregående sektion så skulle ingenting skrivas ut om vi satte variabeln `x` till 9. Då är det möjligt att lägga till en `else`-sats som körs när inget villkor i stystrukturen är sant.

```
x = 9;

if (x > 10) {
    print("Start");
} else if (x > 20) {
    print("Branch 1");
} else if (x > 30) {
    print("Branch 2");
} else if (x > 40) {
    print("Branch 3");
} else if (x > 50) {
    print("Branch 4");
} else {
    print("Default");
}

==> Default
```

Listing 21: En styrstruktur med en else-sats.

Eftersom att inga av villkoren utvärderades till sanna kommer kodexemplet i Listing 21 att köra `else`-blocket, i detta fall kommer programmet att printa ut `"Default"`. `else`-satser inom programmering används oftast för att ge styrstrukturer ett värde för att fånga de allmänna fallen, där ingen explicit jämförelse är sann.

5.2 Loopar

Inom programmering är det vanligt att vilja upprepa vissa block med kod ett specifikt antal gånger. Detta åstadkommer man genom att skapa en loop, som kan upprepa ett block av kod tills dess att ett specifikt villkor har uppfyllts. I SproutScript finns det tre huvudsakliga varianter av loopar.

5.2.1 While-loop

En `while`-loop kan upprepa ett block med kod ett godtyckligt antal gånger tills dess att ett villkor har uppfyllts. Betrakta den följande programkoden i listing 22:

```
counter = 0;

while (counter < 2) {
    print(counter);
    counter += 1;
}

while (counter != 2) {
    print("We won't enter this loop!");
}

==> 0
==> 1
```

Listing 22: En enkel while-loop.

För att skapa en `while`-loop så behöver vi förse loopen med ett villkor. Innan loopen körs kommer detta villkor att kontrolleras, och koden i blocket – det som ligger inom klammerparenteserna – kommer endast att exekveras om villkoret är sant. Så snart som villkoret blir falskt kommer loopen att avbrytas. I listing 22 ser vi att variabeln `counter` deklareras med värdet 0, och detta värde ökar sedan med 1 varje varv i loopen. När loopen når slutet av blocket kommer villkoret att utvärderas ännu en gång. Det är endast om villkoret fortfarande är sant som blocket kommer att upprepas – vilket innebär att loopen avbryts när värdet på variabeln når 2.

Det är fullt möjligt att skapa en loop som aldrig avbryts, och detta är något som man bör vara uppmärksam på när man skriver kod. I det följande kodexemplet skapar vi en loop som aldrig kommer att avbrytas, då villkoret alltid kommer att vara sant.

```
counter = 0;
while (counter < 2) {
    print(counter);
}

==> 0
==> 0
// ...
```

Listing 23: En oändlig while-loop.

Eftersom värdet på variabeln `counter` aldrig uppdateras i listing 23 så kommer loopen att fortsätta i all oändlighet. Detta beror på att värdet på jämförelseoperatorn alltid är sann. I praktiken innebär detta att man skulle kunna ha det booleska värdet `True` som villkor för att få en oändlig loop.

Dock kan man alltid avbryta en loop manuellt, genom att använda nyckelordet **`break`**. När detta nyckelord skrivs i en loop (som i listing 24) så kommer programmet omedelbart att lämna loopen och gå vidare till nästa rad kod, oavsett om villkoret är sant eller falskt.

```
counter = 2;
while (True) {
    counter = counter * 2;
    print(counter);
    if (counter == 16) {
        break;
    }
}

==> 4
==> 8
==> 16
```

Listing 24: Loopen bryts när värdet på variabeln `counter` blir 16.

5.2.2 Do While-loop

En **`do while`**-loop liknar en **`while`**-loop med en viktig skillnad: koden i blocket kommer att exekveras minst en gång, då villkoret kontrolleras först efter att koden har utförts en gång. Här är ett exempel på en **`do while`**-loop:

```
counter = 0;
do {
    print(counter);
} while (counter < 0);

==> 0
```

Listing 25: En enkel do while-loop.

Koden i listing 25 att köras minst en gång, eftersom villkoret `counter < 0` kontrolleras först efter att koden i blocket har utförts. Trots att jämförelsen

är falsk vid första körningen kommer minst en iteration alltid att köras innan villkoret kontrolleras. Är villkoret sant efter första iterationen kommer loopen att upprepas.

5.2.3 For-loop

En `for`-loop är en annan typ av loop som är särskilt användbar när man vill upprepa en kod ett bestämt antal gånger. En `for`-loop har tre komponenter: initiering av en styrvariabel, utvärdering av ett villkor med denna styrvariabel samt någon form av uppdatering av styrvariabeln.

```
for (i = 0; i < 2; i++) {
    print(i);
}

==> 0
==> 1
```

Listing 26: En enkel for-loop.

I exemplet från listing 26 initieras en variabel `i` med värdet 0, villkoret `i < 2` kontrolleras och variabeln uppdateras med `i++` för varje iteration. När värdet på variabeln når 2 så avslutas loopen. Det är inte strikt nödvändigt att inkrementera en variabel med `i++`, utan det går att använda ett aritmetiskt uttryck såsom `i += 2` för att öka styrvariabeln med två. Alternativt kan man initiera styrvariabeln med 10, och minska värdet på den med `i -= 1` för att gå nedåt i loopen. Säkerställ bara att villkoret kan bli sant, annars kommer loopen att vara oändlig.

5.3 Scope

Scope är ett begrepp som beskriver synligheten och livslängden för variabler och funktioner i en programkod. På svenska kallas detta för räckvidd. Beroende på vilket programmeringsspråk man arbetar i kan räckvidden för variabler variera.

I SproutScript så hamnar alla variabler som skapas i den globala räckvidden, medan alla styrstrukturer och loopar får ett eget scope. Skulle en variabel deklareras inuti en loop så kommer denna variabel endast att leva inuti denna loop - något som vi kan analysera i föregående sektion när vi beskrev for-loopar. Här deklarerade vi en styrvariabel - `i` - som användes för att räkna

antalet iterationer. Denna variabel kommer endast att vara åtkomlig inuti denna loop, och så snart som loopen är klar kommer den att raderas. Betrakta följande exempel:

```
for (i = 0; i < 2; i++) {
    print(i);
}

// Utanför loopen kommer vi inte att kunna komma åt variabeln i
print(i); // Denna utskrift kommer att misslyckas!

x = 10;
if (x == 10) {
    y = 20;
}

print(y); // Även denna variabel går inte att komma åt,
// då den deklarerades i styrstrukturen.
```

Listing 27: Ett exempel på en variabel utanför sitt scope.

I kodexemplet från listing 27 så går det inte att använda variabeln `i` utanför loopen, utan denna variabel raderas när blocket kod avslutas. I SproutScript skapar följande strukturer ett nytt scope:

1. Styrstrukturer (såsom **`if`**, **`else if`** och **`else`**)
2. Loopar (såsom **`for`**, **`while`** och **`do while`**)
3. Funktioner, vars räckvidd kommer att beskrivas i sektion sektion 6.

6 Funktioner

Funktioner är kodblock som kan återanvändas och kallas på från olika delar av ett program. De hjälper till att strukturera koden och göra den mer lättläst och underhållbar. Om man finner behov av att ofta upprepa stycken med kod för att exempelvis göra en matematisk uträkning eller någon mer komplicerad kontroll av en variabel är det ofta onödigt att skriva den upprepade gånger i programkoden. Istället kan denna kod användas som en del av en funktion, och således köras väldigt enkelt genom att kalla på denna funktion.

6.1 Definiera en funktion

För att definiera en funktion används nyckelordet `function` följt av ett funktionsnamn och en parameterlista inom parenteser. Koden för funktionen skrivs inom klammerparenteser som följer parameterlistan.

Betrakta följande exempel:

```
function say_this(word) {
    print(word);
}

say_this("Hello!);

==> Hello!
```

Listing 28: En enkel funktionsdefinition.

Listing 28 visar en ytterst simpel funktion, vars enda syfte är att ta emot en variabel och skriva ut den. I praktiken är detta vad som redan händer när vi använder SproutScripts egna `print`-funktion!

6.1.1 Parametrar

Funktioner har en strikt isolerad räckvidd, vilket innebär att en funktion endast kommer åt de variabler som deklareras inom dess egna scope. Detta innebär att funktioner, till skillnad från loopar och styrstrukturer, inte kan komma åt variabler som deklarerats i det globala scopet. För att skicka med en variabel från det globala scopet behöver man skicka med dessa som parametrar.

Parametrar är variabler som skickas till funktion när den anropas. Dessa parametrar används för att skicka information mellan olika delar av programmet och anpassa funktionens beteende baserat på de värden som skickas in. Parametrarna definieras inom parenteserna efter funktionsnamnet, och flera parametrar separeras med kommatecken.

```
function add(a, b) { print(a + b); }
add(2, 3);
==> 5
```

Listing 29: En funktion som tar två parametrar och skriver ut summan.

I listing 29 tar funktionen `add` två parametrar, `a` och `b`, och returnerar deras summa. När funktionen sedan ska kallas på måste antalet parametrar i funktionskallet exakt motsvara antalet parametrar i funktionsdefinitionen.

6.1.2 Returvärde

En funktion kan returnera ett värde till den del av koden som anropade funktionen. För att göra detta används nyckelordet `return` följt av det värde som ska returneras. Detta görs genom att skriva variabeln som skall returneras efter nyckelordet `return`.

```
function multiply(a, b) {
    return a * b;
}
```

Listing 30: En funktion som tar två parametrar och returnerar deras produkt.

I exemplet från listing 30 tar funktionen `multiply` emot två parametrar, `a` och `b`, och returnerar deras produkt. Om data som bearbetas i en funktion behöver användas utanför funktionen är det således viktigt att definiera ett returvärde.

En funktion måste inte ha ett returvärde. För en funktion som saknar nyckelordet `return` kommer returvärdet per automatik vara `nil` – alltså ett odefinierat värde.

6.2 Kalla på en funktion

För att använda en funktion i koden måste den anropas. Funktionen anropas genom att skriva dess namn följt av en lista med argument inom parenteser. Argumenten motsvarar parametrarna som definierades när funktionen skapades. Antalet parametrar måste exakt matcha antalet argument som angavs när funktionen definierades.

```
sum = add(3, 4);
print(sum);
==> 7
product = multiply(3, 4);
print(product);
==> 12
```

Listing 31: Anropa funktioner och använda deras returvärden.

I listing 31 anropas funktionerna `add` (som definierades i listing 29) och `multiply` (som definierades i listing 30) med argumenten 3 och 4. Funktionernas returvärden tilldelas variablerna `sum` och `product`, som sedan skrivs ut.

Notera att det är möjligt att lagra resultatet från ett funktionskall i en variabel. På samma sätt är det även möjligt att använda ett funktionskall för att skicka in en parameter till en annan funktion eller som en del av en ekvation. Ett exempel på detta ser vi i sektion 6.3.

6.3 Rekursion

Rekursion kan användas för att lösa problem där en uppgift kan brytas ner i mindre, liknande uppgifter. I praktiken så definieras en funktion som kommer att kalla på sig själv som en del av sin kod, vilket skapar en form av loop – för varje steg i denna loop kommer funktionen att köras igen, och returvärdet funktionen kommer därefter att vara det samlade resultatet för hela den rekursiva körningen. Ett klassiskt exempel är beräkning av fakultet för ett heltal (n!). Fakulteten för ett tal är produkten av alla heltal från 1 upp till talet självt. Fakultetsfunktionen kan definieras rekursivt enligt följande logik:

- Om n = 0 eller n = 1, är n! = 1.
- Om n > 1, är n! = n * (n-1)!, där (n-1)! beräknas genom att anropa fakultetsfunktionen med (n-1) som argument.

I listing 32 ser vi ett exempel på hur det går att implementera en rekursiv fakultetsfunktion i SproutScript:

```
function factorial(n) {
    if (n == 1 or n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

print(factorial(5));

==> 120
```

Listing 32: Implementation av en rekursiv fakultetsfunktion i SproutScript.

7 SproutScripts övriga funktioner

Under denna dokumentation har vi redan stött på några av SproutScripts egna funktioner, exempelvis `print`. Denna sektion täcker de inbyggda funktioner i SproutScript som möjliggör skapandet av kompletta program.

7.1 Kommentarer

I flera kodexempel under dokumentationens har det skrivits kommentarer i koden för att ge mer kontext – detta har gjorts på samma sätt som SproutScripts kommentarer hanteras. När du skriver två snedstreck i ditt program så kommer allting som står efter dessa snedstreck på samma rad att ignoreras av programmet, och detta kommer inte att exekveras eller tolkas när programmet körs.

```
// En kommentar skrivs på detta sätt!
```

Detta är användbart när du vill skriva anteckningar i din kod, alternativt om du vill skriva beskrivande texter som kan förklara vad ett stycke kod gör. Det är god praxis att skriva goda kommentarer som förklarar lite mer komplexa stycken kod, samt att beskriva syftet med en funktion med en kommentar.

7.2 Input

Funktionen `input` låter en användare av ett SproutScript-program skriva in en text, som därefter kan sparas undan till en variabel. Med denna funktionalitet går det att skapa interaktivitet med användaren genom att ta emot dennes text till ett färdigt program.

```
message = input("Vad heter du? ");

==> Vad heter du? Love
// Notera att "Love" skrevs in av användaren!

print("Hej " + message + "!");

==> Hej Love!
```

Listing 33: Användarens namn hämtas in.

Strängen som skickas in till `input` i listing 33 skrivs direkt ut, och användaren

får sedan skriva in sin text. När användaren trycker på enter-knappen så returnerar `input`-funktionen den sträng som knappats in.

7.3 Klassidentifikaton

Ibland är det nyttigt att kunna kontrollera vad för datatyp en variabel är. Ibland kan man glömma bort, och ibland kan man råka skicka in fel datatyp till en funktion – exempelvis blir det svårt om man skickar in en sträng till en funktion som ska dividera tal. Därför finns det en funktion som kan kontrollera vad för datatyp en viss variabel har.

Denna funktion heter `what_is`, och returnerar en sträng som beskriver vilken datatyp en viss variabel har. Eftersom funktionen returnerar en sträng kan den användas i styrstrukturer för att ändra vilken operation som körs om en variabel är av fel klass.

```
variable_one = [1, 2, 3]
variable_two = 10.01
variable_three = "Hello"

print(what_is(variable_one));
==> list
print(what_is(variable_one[0]));
==> integer
print(what_is(variable_two));
==> float
print(what_is(variable_three));
==> string
```

Listing 34: Hur klassidentifikation kan åstadkommas i SproutScript.

7.4 Test

Det är ofta nyttigt att utföra tester inom programmering. Att testa ett program är i sig en hel vetenskap, och det är väldigt fördelaktigt att börja arbeta med tester samtidigt som man lär sig programmeringens grundprinciper. För att underlätta detta så har SproutScript en egen inbyggd test-funktion.

Funktionen `test` accepterar två parametrar - två stycken uttryck som ska jämföras. När programmet har körts klart så kommer resultatet av samtliga tester att sammanställas och skrivas ut i terminalen. Ett test anses lyckat

om de två värden som skickas in i funktionen är ekvivalenta - alltså att de har samma värde. Se listing 35 för ett exempel på hur detta används, och vilken utdata som test-funktionen ger.

```
function factorial(n) {
    if (n == 0 or n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

test(factorial(5), 120);
test(factorial(8), 40320);

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-

Program finished! Commencing tests...
Test 0          |        Testing if 120 == 120:        PASSED!
Test 1          |        Testing if 40320 == 40320:    PASSED!

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
All tests were successful!
```

Listing 35: Testning av en rekursiv fakultetsfunktion i SproutScript.

8 SproutScript – språkreferens

Här återfinns en snabb referenspunkt för den funktionalitet som SproutScript erbjuder.

Tabell 6: En sammanfattning av grundläggande operationer i SproutScript

Kategori	**Beskrivning**	**Exempel**
Variabeldeklaration	Deklarera en variabel	`x = "Valfri data";`

Datatyper
int
float
string
boolean
array
	
	Heltal
	Flyttal
	Textsträng
	Booleskt uttryck
	Samling av element
		
		`x = 42;`
		`y = 3.14f;`
		`s = "Hello";`
		`b = True;`
		`a = [1, 2, 3];`

Aritmetiska operationer
+
-
*
/
%
	
	Addition
	Subtraktion
	Multiplikation
	Division
	Modulus
		
		`x = a + b;`
		`x = a - b;`
		`x = a * b;`
		`x = a / b;`
		`x = a % b;`

Logiska operationer
&&
||
!
	
	Logiskt AND
	Logiskt OR
	Logiskt NOT
		
		`if (a and b)`
		`if (a or b)`
		`if (!a)`

Jämförelseoperatorer
==
!=
<
>
<=
>=
	
	Lika med
	Inte lika med
	Mindre än
	Större än
	Mindre än eller lika med
	Större än eller lika med
		
		`if (a == b)`
		`if (a != b)`
		`if (a < b)`
		`if (a > b)`
		`if (a <= b)`
		`if (a >= b)`

Styrstrukturer
if
else
 	
else if
	
	Villkorligt utförande
	Alternativt utförande
	
	Kedja av villkor
		
		`if (a > b)`
		`if (a > b) { ... } else { ... }`
		`if (a > b) { ... } else if (a < b) { ... }`

Tabell 7: En sammanfattning av grundläggande operationer i SproutScript

Loopar		
for	Upprepa kodblock ett visst antal gånger	`for (i = 0; i < 10; i++) { ... }`
while	Upprepa kodblock så länge villkor är sant	`while (a < b) { ... }`
do-while	Upprepa kodblock minst en gång, så länge villkor är sant	`do { ... } while (a < b);`
Funktionsdefinition	Definiera en funktion	`my_function(a, b) { ... }`
Funktionsanrop	Anropa en funktion	`my_function(3, 5);`
print(variable)	Skriver ut meddelande till terminalen	`print(x);`
input(message)	Låter användaren skriva in text	`input("Enter your name: ");`
what_is(variable)	Ger en sträng med datatypen	`what_is(x);`
test(var1, var2)	Testar om variabler är ekvivalenta	`test(x, y);`

9 Kodexempel

Denna sektion visar två enklare programmeringsövningar – Fizz Buzz i listing 36, och Bubble Sort i listing 37 – som de implementeras i SproutScript.

```
for (i = 1; i <= 100; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
        print("Fizz Buzz");
    } else if (i % 3 == 0) {
       print("Fizz");
    } else if (i % 5 == 0) {
       print("Buzz");
    } else {
        print(i);
    }
}
```

Listing 36: Fizz Buzz - En vanlig programmeringsövning.

```
arr = [800, 11, 50, 771, 649, 770, 240, 9];
print(arr);
print("Sorting...");

for (write = 0; write < length(arr); write++) {
    for (sort = 0; sort < length(arr) - 1; sort++) {
        if (arr[sort] > arr[sort + 1]) {
            temp = arr[sort + 1];
            arr[sort + 1] = arr[sort];
            arr[sort] = temp;
        }
    }
}

print(arr);
```

Listing 37: En implementation av Bubble Sort i SproutScript.

SproutScript - Systemdokumentation

I denna sektion beskriver vi hur SproutScript fungerar i praktiken, hur den användarskrivna koden tolkas och vad som sker i bakgrunden när SproutScript exekveras.

10 Språkets uppbyggnad

SproutScript är ett interpreterat språk - all kod kommer således att tolkas och exekveras direkt, snarare än att generera separata filer med tolkad programkod. Hur en fil tolkas, och hur ett program därefter byggs upp utifrån detta kommer att täckas i detalj under de nästkommande underrubrikerna, men sammanfattningsvis sker följande steg när ett SproutScript-program körs:

1. Filen öppnas av `sproutscript.rb`. Här sker kontroller av filändelser, samt tolkning av kommandoradsargument.
2. Hela SproutScript-filens innehåll sammanfattas i en sträng, som därefter skickas till parsern.
3. Parsern `rdparse.rb` tillhandahölls av kursledningen för TDP019 och användes oförändrad. Denna parser tillät oss att definiera tokens och regler, som därefter nyttjades för att konstruera ett syntax-träd med noder för exekvering.
4. Strängen som skickades till `rdparse` läses tecken för tecken, och parsern försöker därefter att matcha varje tecken med en token.
5. När hela filen lästs, och tokens genererats, kommer dessa tokens i ordning att matcha SproutScripts regler.
6. Exakt funktionalitet skiljer sig från regel till regel, men varje regel kommer att generera en programnod som lagras i ett syntax-träd. Detta innebär att alla tokens i den parsade filen skall generera en eller flera programnoder innan någon av den användardefinierade koden exekveras.
7. Efter att parsern har skapat hela syntax-trädet, så kommer `sproutscript.rb` att använda funktionen `run` på det parseobjekt som genererades vid parsningssteget. Denna funktion kommer att iterera genom samtliga noder i syntax-trädet, och köra varje nods run-funktion.
8. Genom detta exekveras koden sedan steg för steg.

Filorganisationen är byggd för att vara logisk och lättnavigerad inom en lämplig IDE, och importeras sedan till lämpliga filer för exekvering. Koden är formatterad med hjälp av RuboCop för att ge konsekvent indentering och syntax.

10.1 Parsning

Vi valde att nyttja den i kursen tillhandahållna parsern `rdparse.rb` för SproutScript. Denna parser är en så kallad recursive descent parser[9]. Eftersom vi inte gav oss i kast med att skriva en egen parser kunde vi fokusera mer på att arbeta med det faktiska språket, snarare än de intrikata detaljerna av att konstruera en lexer och parser.

`rdparse.rb` behöver dels tokens, som definieras i form av RegEx-uttryck som kan användas i sin lexer, samt regler som definierar vad som skall ske när dessa tokens hittas i en specifik ordning. När en token hittas genererar den för alla operatorer och skyddade nyckelord en symbol, som sedan kan användas för att matcha regler. Därefter matchas strängar och siffror som inte matchar någon annan tokens till just rena strängar, då detta är användardefinierad text, exempelvis variabelnamn.

De regler som dessa tokens därefter matchar definieras i detalj i sektion 12. Gemensamt för samtliga regler är att de genererar programnoder som på egen hand inte utför några operationer, utan endast registrerar de aktiviteter som skall utföras samt i vilken ordning.

I parsningssteget skapas även objekt av klassen `Frame`, som definierar vilket scope som den nuvarande delen av programkoden befinner sig. Nästintill samtliga objekt har detta objekt som en parameter vid instansiering så att relevanta objekt har möjlighet att komma åt variabler som lagras i den nuvarande framen. När en del av koden skapar ett nytt scope kommer detta att lagras i en instansvariabel för parsern, så att det scope - eller den frame - som bifogas till vardera nod alltid är den korrekta för det nuvarande steget i parsningen.

Efter att parsningen är genomförd kommer parsern att ha skapat ett `SproutParser`-objekt. Detta objekt har en `run`-funktion som exekverar samtliga noder i sektion 10.3.

[9] `https://en.wikipedia.org/wiki/Recursive_descent_parser`

10.1.1 BNF

Reglerna för hur text matchar i vår parser är noterat i dokumentet BNF i sektion 13, bilagor. Detta dokument visar de logiska regler som användarskriven kod matchar enligt Backus-Naur form[10], eller BNF.

10.2 Nodstruktur

Under parsningssteget skapas det individuella noder som motsvarar varje enskild händelse i programkoden. Händelser såsom en variabeltilldeling, en aritmetisk operation eller en loop genererar alla en programnod, där varje nod går att exekvera individuellt. Oavsett om det är en nod som utför en större operation - exempelvis en aritmetisk operation - eller en enklare datanod såsom en integer, så har varje nod en dedikerad nod för körning - `run`.

Då varje nod har en uniform funktion för körning är det enkelt att nästla lämpliga noder inuti varandra. Aritmetiska operationer är ett utmärkt exempel på hur detta nyttjas till vår fördel. En aritmetisk nod har två noder som kombineras med en operator, och bägge led i den aritmetiska operationen kan vara en ytterligare aritmetisk nod. Detta ger stöd för att skapa längre ekvationer, då individuella noder kan nästlas. Detta går att observera i listing 37 där aritmetiknoderna flitigt nästlas och utvärderas till -25 när programmet exekveras.

```
var = 200 / 2 / 2 / -2;
print(var);

==> -25
```

Listing 38: Nästlade aritmetiknoder

10.3 Exekvering av noder

Eftersom samtliga noder skapas i parsningssteget så sker inga operationer innan hela filen har gått igenom parsningssteget. Detta innebär att användarens kod inte kommer att utföra någon operation innan `SproutParser`-objektet väl körs.

När denna huvudnod körs, så kommer `SproutParser`-objektet att köra varje

[10] `https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form`

individuell nod i syntaxträdet i den ordning de skapats. Varje individuell nod i syntax träder kommer att exekveras med `run`. Alla noder som utvärderas kommer att ha egna överlagrade run funktioner från diverse basklasser i språket. Ett bra exempel på en nod som oftast behöver i flera steg utvärderas är aritmetik noden. Vid beräkningar kan det förekomma att flertalet aritmetik noder och integer noder behöver utvärderas.

```
lhs_current = get_data(@lhs)
rhs_current = get_data(@rhs)

if lhs_current.instance_of? Integer
and rhs_current.instance_of? Integer
  SproutInt.new(lhs_current.send(@op.to_sym, rhs_current))
elsif lhs_current.instance_of? Float
or rhs_current.instance_of? Float
  SproutFloat.new(lhs_current.send(@op.to_sym, rhs_current))
end
```

Listing 39: Aritmetik-nodens run funtkion

I listing 38 så använder vi `get_data` (se listing 41 för antingen hämta variabler, alternativt bryta ned en SproutScript-nod till ett värde Ruby kan tolka. Är det exempelvis en aritmetisk nod i ett av leden kommer denna utvärderas, och resultatet av denna kommer att skickas tillbaka från `run` Därefter utvärderas uträkningens vänster- och högerled med hjälp av Rubys `send`-metod.

10.4 Scopehantering

Scopehanteringen i SproutScript nyttjar en egenskriven klass - Frame. Denna klass lagrar alla variabler som skapats i ett visst scope. Se sektion sektion 5 för detaljerad info kring när nya scopes skapas. I listing 40 kan vi se ett exempel på hur en användardefinierad funktions scope fungerar. Dessa kommer att skapa ett nytt Frame-objekt, vilket gör att detta blir ett nytt lokalt scope för funktionen som inte är kopplat till det globala scopet.

För att styrstrukturer och loopar skulle kunna skapa ett eget scope, men även kunna kontrollera det globala scopet så har varje Frame-objekt en referens till sitt föregående scope i medlemsvariabeln `@parent`. En rekursiv kontroll av denna medlemsvariabel tillät oss att kontrollera alla nästlade scopes för att hitta en variabel med medlemsfunktionen `get_data`, som visas i sin helhet i

```
new_value = 10; // Globalt scope

function foo(){ // Lokalt scope skapas
    print(new_value); // Den globala variabeln kan ej hämtas
}
```

Listing 40: Globalt kontra lokalt scope

listing 41. Denna hantering möjliggjorde djup nästling av styrstrukturer likväl som loopar.

```
 def get_var(var)
    return @vars[var.to_sym] if @vars.key?(var.to_sym)
    return @parent.get_var(var) unless
@parent.instance_of? NilClass

    raise SproutVariableError, var
end
```

Listing 41: Frame-objektets `get_data`-funktion

11 Klasshierarki

Tydlig klasshierarki implementerades i språket tidigt, vilket möjliggjorde enkla jämförelser av basklasser under exekvering och parsning. Gemensamt för alla noder är att grundläggande datanoder ärver från `SproutMainData`, samt att alla operatorer såsom aritmetiska operationer, logiska operationer samt jämförelser ärver från basklassen `Node`. Detta gör det simpelt att kontrollera i parsning likväl som exekvering huruvida en nod härleder från en viss basklass, genom att exempelvis kasta felmeddelanden:
`raise SproutValueError unless node.is_a? Node`.

Samma grundläggande princip nyttjades för SproutScripts inbyggda funktioner, där dessa ärver från basklassen `SproutFunctions`. Vidare, medan det inte nyttjats praktiskt så ärver även samtliga loopar från `SproutLoop`. Avstickaren från denna praxis är samtliga noder som beror användardefinierade funktioner, som inte har någon arvsstruktur – men detta beror på att dessa inte interagerar med varandra på samma sätt, utan dessa är unika noder. Att definiera en funktion, kalla på en funktion och returnera ett värde från en

funktion har i SproutScripts nodstruktur ytterst lite gemensamt, vilket fick oss att anse det överflödigt att skapa en arvsstruktur för dessa då de behöver kontrolleras individuellt.

12 Analys och reflektion

Inledningsvis i detta projekt var ambitionsnivån hög. Konceptet för SproutScript var att skapa ett interpreterat språk som kombinerade de användarvänliga aspekterna från språk som Python och Ruby med den striktare syntax som finns i språk som C, C++ och i viss mån JavaScript. De enklare aspekter som implementerades var dynamisk typning och enklare funktionsanrop (alltså `print()` för utskrift snarare än `Console.Write();`, medan att nyttja semikolon som radändelse samt klammerparenterser för att definiera block adopterats från C-baserade språk. Vidare så fanns ambitionen att skapa tydlig och hjälpsam felhantering, så att det för en användare inte framgick att SproutScript var byggt ovanpå Ruby – alltså skulle Rubys egna felmeddelanden och stacktraces gömmas bakom egen implementation.

12.1 Saknad funktionalitet

All planerad funktionalitet implementerades i språket med två undantag. Användardefinierade klasser med arv implementerades inte, och den felhantering som byggts för SproutScript är undermålig kontra den vision som fanns vid projektets start. Utöver dessa två aspekter har däremot all planerad funktionalitet implementerats.

Klasser var en planerad funktionalitet som vi övergav på grund av tidsbrist. Medan vi hade planerat projektet utefter bästa förmåga allokerades för lite tid åt andra kurser under projektets gång. Kursen som lästes parallellt med projektet under hösten krävde mer tid än vad som ursprungligen uppskattades, något som påverkade de sista funktionerna som planerats. Medan bristen på klasser i SproutScript är beklagligt så anser vi detta till trots att språket är brukligt med den funktionalitet som väl implementerats.

Felhanteringen i SproutScript är delvis implementerad, däremot har ambitionsnivån inte uppnåtts till fullo. I den mån det är möjligt har samtliga fel som upptäckts under konstruktionen av språket associerats med egenskrivna felmeddelanden, däremot var ambitionen att även skriva egna stacktraces som hänvisade användaren till den rad i deras egna kod som orsakade felet. Detta visade sig vara ett komplicerat problem som vi inte hade tid att lösa.

12.2 Utmaningar

Under konstruktionen av SproutScript stötte vi på ett antal utmaningar - däremot inga som var oöverkomliga. En av de större utmaningarna som krävde större omskrivning av befintlig kod var funktioner med rekursion. Medan det ursprungliga konceptet för funktionsdefinitioner och funktionskall fungerade utmärkt orsakade parsningen av användarens kod för dessa funktioner viss patrull. Problematiken var till stor del faktumet att våra tokens i parsern vid detta tillfälle var rena strängar, snarare än symboler. Detta i kombination med logiska brister i parsern vid detta tillfälle gjorde att markant fler rader kod än tänkt matchade regeln för funktionskall, något som bröt funktionaliteten för andra delar av koden.

En annan utmaning var nästlade loopar. Att säkerställa att rätt iteration av loopen kördes varje varv, och att styrvariabeln återställdes visade sig vara en stor utmaning för samtliga loopar vi konstruerade. Problematiken, som vi upptäckte i efterhand, var att jämförelser likväl som aritmetiska operationer ändrade på den variabel som lagrades i scopet, vilket gjorde att loopen efter en genomförd iteration redan hade sitt villkor uppfyllt. Lösningen på detta problem var att skapa djupa kopior av elementen i fråga, samt att säkerställa att dessa kopior lagrades i temporära variabler som sedan återställdes vid nästa körning. Detta hjälpte oss även att bygga upp en mer stabil och konsekvent styrstruktur som klarade av att upprepas.

Returvärden för funktioner var även ett markant problem under utvecklingen. Under en stor del av utvecklingen hade SproutScripts `return`-nod inte möjlighet att nästlas i styrstrukturer eller loopar inuti en funktionsdefinition. Detta ledde till att kod behövde skrivas på ett sätt som är långt från intuitivt. Problemet med detta var att retur-noden, när den kördes i en sturstruktur till exempel, hade som grundfunktionalitet att skicka tillbaka sitt värde från denna nod. Styrstrukturernas noder hade däremot ingen inbyggd funktionalitet att skicka tillbaka ett värde. Lösningen på detta var att loopar likväl som styrstrukturer kontrollerar huruvida en nod i dess syntax-träd är en retur-nod, och returnerar noden om så är fallet. Funktionerna lagrar sedan returvärdet från alla noder i en variabel, som kontrolleras för att identifiera en eventuell retur-nod.

12.3 Lärdomar

Detta projekt gav oss många nyttiga lärdomar, där mycket av det kretsade kring projektadministration och att anpassa vår definition av en färdigställd produkt till den givna deadlinen. Dock så är de främsta lärdomarna från denna

kurs starkt knutna till exekvering och parsning av användarskriven kod. De största lärdomarna inom detta fält är som följer:

- **Förståelse för språkdesign:** Genom att skapa vårt eget programmeringsspråk fick vi en djupare förståelse för hur språk är designade och hur olika komponenter samverkar. Detta innefattade bland annat hur syntax, semantik och grammatik fungerar samt hur språkets struktur och konstruktioner är uppbyggda.

- **Lexer och parser:** Implementering av ett programmeringsspråk krävde att vi brukade en lexer och parser, samt att vi byggde upp en god förståelse för hur dessa arbetar. Lexerm omvandlade den inmatade källkoden till en sekvens av tokens, medan parserm analyserade och tolkade dessa tokens för att bygga upp en intern representation av koden. Genom att arbeta nära dessa komponenter förstod vi bättre hur källkod tolkas och hur olika språkelement representeras internt – och denna kunskap är starkt kopplad till förståelsen för abstrakta syntaxträd.

- **Abstract Syntax Tree (AST):** Ett vanligt sätt att representera den interna strukturen av ett programmeringsspråk är genom en abstrakt syntaxträd (AST), något som vi implementerat från grunden upp i SproutScript. Vårt interna AST var starkt modellerat efter de exempel som täcktes under föreläsningarna i den föregående kursen som var förberedande inför detta projekt. Genom att skapa och manipulera vårt eget AST för SproutScript fick vi praktisk erfarenhet av hur kod struktureras och tolkas av kompilatorer eller en interpretator – en lärdom som vi kan komma att dra stor nytta av i framtiden.

- **Runtime environment:** Att skapa och definiera en körmiljö för vårt programmeringsspråk gav djupare kunskap och förståelse för hur program exekveras, samt hur resurser allokeras och bearbetas under programmets körning. Det innefattade bland annat hantering av variabler, funktioner, minnesallokering och eventuell felhantering.

- **Felsökning och optimering:** Som noterat i sektion 12.2 stötte vi på ett flertal större utmaningar och buggar under konstruktionen av SproutScript, vilket gav oss god möjlighet att förbättra våra felsökningsfärdigheter. Mycket tid under projektet har gått till att analysera den befintliga koden för att identifiera om problemet vi bearbetade berodde på ett logiskt fel i parsern, felaktiga eller felallokerade parametrar vid konstruktion av objekt, eller om det berodde på mer fundamentala fel med det tillvägagångssätt vi implementerat.

- **Testning:** Sättet som SproutScript har testats på var genom att skriva programkod, och skicka in den i parsern för att se om resultatet blev korrekt. Med facit i hand bör vi vid ett mycket tidigare skede nyttjat en testmodul såsom Rubys `unit:test` för att skriva tester av varje enskild modul. Medan det finns tester skrivna med denna modul så bör de vara markant mycket noggrannare med mer hänsyn taget till fler scenarion. Lärdomen här är att det är viktigt att arbeta betydligt mycket mer proaktivt med tester, snarare än reaktivt efter att någonting inte fungerat vid ett senare steg i programmeringen.

13 Bilagor

SproutScript BNF