

TDP019 Projekt: Datorspråk

Dopo Språkdokumentation

Författare

Kasper Nilsson, kasni325@student.liu.se
Simon Gradin, simgr011@student.liu.se

Abstract

Rapporten dokumenterar skapandet och användandet av det egenskapande imperativa programmeringsspråket Dopo. Målet var att skapa ett språk med minimal och explicit syntax som lätt kan plockas upp av nybörjare såväl som erfarna programmerare. Språket skrevs under en period på ca 3 månader av studenter på kandidatprogrammet innovativ programmering vid Linköping Universitet. Resultatet av projektet blev ett språk som följer det imperativa programmeringsparadigmen med olika satser för repetition och selektion för att förändra flödet för ett program. En konsekvens av val av syntax är att kedjning av flera uttryck med operatorer är besvärligt samt att det språket har en viss fördröjning vid körning.

Innehållsförteckning

1 Inledning	1
2 Användarhandledning.....	2
2.1 Installation och användning.....	2
2.1.1 Nedladdningsplats	2
2.1.2 Mjukvarukrav	2
2.1.3 Körning via filer	3
2.1.4 Interaktiv körning.....	3
2.2 Grundläggande datatyper	4
2.2.1 String_c	4
2.2.2 Number_c.....	4
2.2.3 Bool_c.....	4
2.2.4 Nil_c	5
2.3 Sammansatta datatyper.....	5
2.3.1 List.....	5
2.3.2 Dictionary	5
2.4 Block.....	6
2.5 Räckvidd	6
2.6 Tilldelning.....	7
2.6.1 Variabler.....	7
2.6.2 Funktioner	8
2.7 Funktionkall.....	8
2.8 Selektion-satser.....	9
2.8.1 If-satser	9
2.9 Repetition-satser.....	10
2.9.1 While-loopen.....	10
2.9.2 For each-loopen	10
2.10 Kontrollsatser	11
2.10.1 Abort	11
2.10.2 Continue.....	11
2.10.3 Return	12
2.11 Reserverade namn	12
2.11.1 p	12
2.11.2 g.....	12
2.12 Stdlib	12

2.12.1 Funktionslista	13
2.13 Operatorer	13
2.13.1 Op: +	14
2.13.2 Op: -	14
2.13.3 Op: *	14
2.13.4 Op: /	14
2.13.5 Op: %	14
2.13.6 Op: ^	15
2.13.7 Op: >=	15
2.13.8 Op: <=	15
2.13.9 Op: <	15
2.13.10 Op: >	16
2.13.11 Op: =	16
2.13.12 Op: !=	16
2.13.13 Op: !	17
2.13.14 Op: 	17
2.13.15 Op: &	17
2.13.16 Op: ->	17
2.14 Metafunktionlitet	17
2.14.1 Kommentarer	17
3 Systemdokumentation	18
3.1 BNF-grammatik	18
3.2 Lexning/parsning	20
3.3 Exekvering	21
3.4 Nod Klasser	21
3.4.1 Allmänt om klasserna	21
3.4.2 Dopo	21
3.4.3 Valid_list	22
3.4.4 Call	22
3.4.5 Operator_expr	22
3.5 Algoritmer	23
3.6 Kodstandard	23
3.6.1 Indentering	23
3.6.2 Kommentarer	23
3.6.3 Klassnamn	23

3.6.4 Variabelnamn	23
3.6.5 Funktionsnamn	24
4 Erfarenheter och reflektion	25

1 Inledning

Projektet Dopo är ett programmeringsspråk skrivet i Ruby och skapat under kursen TDP019 – konstruktion av datorspråk. Kursen ingår i kandidatprogrammet innovativ programmering vid Linköping Universitet och läses under termin 2. Kursen läses i par och målen har varit att konstruera ett mindre programmeringsspråk och visa förståelse för hur programmeringsspråk kan vara uppbyggda. Val av språkimplementation är valfri detsamma gäller för vilka verktyg för att uppnå denna vision. För att kunna reflektera över lärandet av projektets olika faser fördes en språkdagbok under projektets gång.

Syftet med Dopo var att skapa ett programmeringsspråk riktat mot en användargrupp med viss programmeringsvana, men även mot nybörjare med grundläggande dator- och terminalvana. Det är ett språk lämpligt för enkla program/script eftersom språket använder sig av en minimal och okonventionell syntax gentemot dagens populära programmeringsspråk. Syntaxen är i många av dess satser likt syntaxen för funktionsanrop för andra populära språk som t.ex. c++ och python3. Skillnaden är att paranteser är omvända och hamnar först, därefter kommer kallet på funktionen eller operationen. För ännu mer djupgående om hur syntaxen är uppbyggd se avsnittet "BNF-grammatik".

Dopo faller inom ett imperativt programmeringsparadigm. Inom detta paradigm löses problem genom att programmeraren använder sig av diverse satser för att besvara **hur** problemen ska lösas. Det finns således satser för selektion, repetition, tilldelning o.s.v. för att förändra ett programs flöde.

2 Användarhandledning

2.1 Installation och användning

I detta avsnitt redogörs vart en användare kan ladda ned Dopo och vilka förutsättningar som krävs för att en användare ska kunna använda språket. Även presenteras hur användaren kan köra Dopo-kod med hjälp av filer eller interaktivt i terminalen.

2.1.1 Nedladdningsplats

Dopo kan hämtas via följande länk:

- <https://gitlab.liu.se/kasni325/dopo-version-1.0>

Länken leder till en första sida över projektets mapp. För att ladda ned Dopo klicka på knappen som visas på *Figur 1* som lokaliseras högt upp i det högra hörnet på hemsidan. Välj önskad komprimeringsformat och extrahera filerna till valfri mapp.



Figur 1 Hämtningsknapp

2.1.2 Mjukvarukrav

För att kunna använda Dopo krävs det att programmeringsspråket Ruby 3.0.1 eller senare version är installerad. För att kontrollera ifall Ruby är installerad skriv in följande inringande kommando enligt *Figur 2* för Windows- och Linux-baserade operativsystem.

```
C:\Users\  
ruby 3.0.1p64
```

(The command 'ruby -v' is circled in red in the original image.)

Figur 2 Kommando för kontroll att Ruby är installerat

Om Ruby inte är installerad följ installationsguiden som finns på följande länk:

- <https://www.ruby-lang.org/en/downloads/>

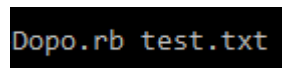
En valfri texteditor rekommenderas för att skriva kod. Några exempel på editors är emacs, VS code, Notepad++ och Atom.

2.1.3 Körning via filer

Givet att stegen ovan är uppföljda kan det börjas skriva kod i Dopo. För att genomföra detta skapa en ny textfil i valfri editor som sparas i samma mapp som "Dopo.rb" finns placerad (bör vara under `din_mapp/tdp019-project/src/`). Öppna upp terminalen och lokalisera i densamma mapp. I den nyskapade textfilen skriv in följande rader:

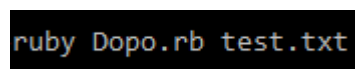
```
()@first_program  
{  
  "Hello World!"  
}  
()  
first_program
```

För att exekvera filen finns två alternativ, *Figur 3* och *Figur 4* anger de två alternativen där "test.txt" är textfilen som innehåller koden ovan. Kommandot som anges i *Figur 3* fungerar enbart för Windows-baserat operativsystem och kommandot som visas i *Figur 5* fungerar enbart för Linux-baserat operativsystem. Kommandot som presenteras i *Figur 4* fungerar för både Linux- och Windows-baserade operativsystem.



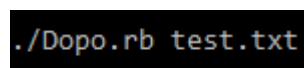
```
Dopo.rb test.txt
```

Figur 3 ett sätt att köra en fil i Dopo. Fungerar bara för Windows.



```
ruby Dopo.rb test.txt
```

Figur 4 ett annat sätt att köra en fil i Dopo. Fungerar för både Windows och Linux.



```
./Dopo.rb test.txt
```

Figur 5 ett annat sätt att köra en fil i Dopo. Fungerar bara för Linux.

Efter något av kommandona som anges i *Figur 3*, *Figur 4* eller *Figur 5* har utförts kommer det att skrivas ut "Hello World" i terminalen.

Observera, för att kunna köra språket på formatet likt *Figur 5* på Linuxsystem kan det behövas göra följande:

1. gå in i mappen där filerna är extraherad
2. kör följande kommando i terminalen: `chmod +x Dopo.rb`

2.1.4 Interaktiv körning

För att exekvera Dopo interaktivt skriv in kommandot i terminalen som anges i *Figur 3*, *Figur 4* eller *Figur 5* men **utelämna** fil efter "Dopo.rb". Observera att interaktivt enbart kör varje rad för sig. Detta innebär att allt som ska köras måste finnas på samma rad i terminalen. I exempel med koden ovan hade det istället sett ut: `()@first_program{ "Hello World!" } ()first_program`

2.2 Grundläggande datatyper

Detta avsnitt presenterar Dopus 4 grundläggande variabel-klasser, `String_c`, `Number_c`, `Bool_c` och `Nil_c`. Dess syntax och egenskaper med tillhörande exempel visas.

2.2.1 `String_c`

Representeras som text omslutet med dubbla citattecken. Text kan innehålla obegränsat med symboler och allt inom citattecken kommer tolkas som en del av strängen.

Till exempel.

```
"foo"
```

```
"b a r"
```

```
".^92rf"
```

Hakklamrar före en sträng kan användas för att hämta element ur strängen. Varje sträng är noll-indexerat.

Operatorer som använder sig av `String_c` kan inte kombineras med andra grundläggande klasser.

2.2.2 `Number_c`

Hanterar både decimaltal samt heltal. Används endast heltal i operationer så utförs heltalsoperationer (heltalsdivision etc.), men används decimaltal i någon del av matematiska operationer så resulterar operationen i ett decimaltal.

Operatorer som använder sig av `Number_c` opererar enligt matematisk aritmetik och kan inte kombineras med andra grundläggande klasser.

Exempel: 3, 3.14, -3.99999

2.2.3 `Bool_c`

Representeras som sant eller falskt värden.

Sant värde skrivs `"T"` eller `"true"`.

Falskt värde skrivs `"F"` eller `"false"`.

Operatorer som använder sig av `Bool_c` värden representerar boolesk algebra och kan inte kombineras med andra grundläggande klasser.

2.2.4 Nil_c

Representerar ett tomt värde.

Skrivs : nil

2.3 Sammansatta datatyper

Detta avsnitt presenterar datatyper som kan innehålla andra datatyper. List samt Dictionary är båda implementerade och kan innehålla alla datatyper inklusive fler List:s och Dictionary:s.

2.3.1 List

Representeras med värden inom hakklammrar separerade med kommatecken. Varje lista är noll-indexerat.

Till exempel:

[1, "foo", T]

Hakklammrar före en lista kan användas för att hämta element ur listan.

Till exempel: [1][1,2]

Returvärdet från exemplet ovan ger 2 från listan.

2.3.2 Dictionary

Representeras som kommaseparerade par av värde och dess nyckel omslutna av hakklammrar. En nyckel kan bestå av samtliga datatyper i språket.

Till exempel:

[1 @ "woo", T @ 69, [] @ "x"] ← Skapar en Dictionary med värde först sedan nyckel i varje par.

Hakklammrar före en Dictionary kan användas för att hämta värden baserat på nyckel

Till exempel:

["woo"] [1@"woo",T@69,[]@"x"] ← Här hämtas värdet 1

2.4 Block

Vid skapning av funktioner, loopar eller if-satser krävs det ett tillhörande block. Varje block kan antingen innehålla flera satser som exekveras i den ordning de är uppsatta i koden eller så kan blocket vara tomt. När blocket är tomt kommer det att returnera ett objekt av klassen Nil_c som i sin tur returnerar Ruby:s egna nil-värde.

För att skapa block måste det finnas en tillhörande funktionsdeklaration, if-sats eller loop. Sedan för att markera blocket används måsvingar ('{', '}'), t.ex.

```
T?  
{                               ← block startar  
  ("Hello world!")p  
}                               ← block slutar
```

2.5 Räckvidd

Räckvidd innebär vart programmeraren har tillgång till tilldelade variabler och funktioner.

I Dopo har samtliga block tillgång till variabler och funktioner som ligger på samma räckviddsnivå eller ovanför sig. När en funktion eller variabel kallas på i programmet kommer det först kolla ifall variabeln eller funktionen finns tilldelad i det block som är längst in nästlad och om det inte finns i detta block kommer sökningen ske stegvis utåt tills vi når det globala blocket.

Det globala blocket är det första som skapas vid varje körning av programmet och kommer alltid vara det yttersta.

2.6 Tilldelning

Detta avsnitt kommer beröra tilldelning av variabler och funktioner i språket Dopo. Vad en parameter är och får vara. Vilka parameteröverföringsmetoder som finns. Vad som krävs för att skapa en tilldelning. En variabel eller funktion måste och har alltid ett värde.

2.6.1 Variabler

Variabler måste ha ett namn som börjar med en stor eller liten bokstav med å, ä och ö inkluderat. Sedan kan namnet bestå av fortsatta stora eller små bokstäver, nummer eller symbolerna \$, £ och _.

Värdet som en variabel kan få kan vara en av de datatyper som anges i avsnittet "grundläggande datatyper", som anges i avsnittet "sammansatta datatyper", matematiskt uttryck, en annan variabels värde eller ett funktionskall.

För att slå upp vilket värde en variabel har kan programmeraren ange i koden antingen (variabel)p som skriver ut värdet i terminalen eller returnera variabeln via retur-operatören (se avsnittet "Return") eller ha den som sista rad i koden.

Några exempel på variabeltilldelning är:

```
(1,x)@    ← x får värdet 1
((1,1)+,x)@ ← x får värdet av resultatet av 1 + 1
(x,y)@    ← y får samma värde som x har
(d)@      ← d får ett default-värde nil
```

Variabler kan även tilldelas med en matematisk operator (+, -, *, /, ^, %) framför @-tecknet. Detta kommer att ersätta värdet för variabeln med nuvarande värdet applicerat med operators operation med den andra parameter. Observera att variabeln redan måste existera innan denna typ av tilldelning kan ske. Exempel på denna typ av tilldelning är:

```
(5,x)@    ← x får värdet 5
(10,x)*@   ← x får värdet av resultatet av 5*10
```

En tilldelning av en variabel har som returvärde variabelns tilldelade värde.

2.6.2 Funktioner

Funktioner består av samma namnregler som för variabler. Skillnaden är att funktioner inte kan ha vissa namn som är reserverade i språket, för mer information kring vilka namn som är reserverade se avsnittet "Reserverade name".

Vidare kan funktioner ha parametrar som är en form av variabler som funktionen enbart har tillgång till i sitt block. När man skapar en funktion kan programmeraren ange dessa parametrar med dess namn, namnen ingår även där i samma namn-regler som för variabler. Parametrar skickas in som kopior som standard parameteröverföring, men en funktion kan ange att den vill att respektive parameter i parameterlistan skickas in som referens. Att en parameter skickas in som referens innebär att variablerna som skickas med i funktionskallet kommer att ändras även utanför funktionens block. Vid parameteröverföring som kopia kommer samma variabler inte att ändras utanför funktionens block.

Några exempel på funktionstilldelning är:

```
(x,y,z)@summa  ← Funktionstilldelning av en funktion som räknar ut summan av 3 tal.  
{  
    Parametrarna skickas in som kopior  
    (x,y,z)+  
}
```

```
(x)@addera_1*  ← Funktionstilldelning av en funktion som adderar 1 till en variabel  
{  
    Parametern skickas in som referens.  
    (1,x)+@  
}
```

Det kan tilldelas funktioner inuti funktionsblock. Då kommer denna tilldelade funktion enbart kunna nås inuti detta funktionsblock samt dess nästlade block som finns inuti.

En tilldelning av en funktion har som returvärde nil.

2.7 Funktionkall

När en funktion är tilldelad så kan programmeraren kalla på funktionen med dess tillhörande namn givet att rätt antal parametrar anges. Vid kallning kommer programmet leta upp funktionen (se avsnittet "Räckvidd") och sedan köra tillhörande block. Funktioner kan även kalla på sig själv, även kallad rekursion.

En funktion kommer alltid ha ett returvärde, det är antingen den sista satsens värde i funktionen som returneras eller kan funktionen returnera tidigare m.h.a. retur-operatorn (<-). När denna operator exekveras avbryts funktionen och värdet som står innan operatorn returneras till platsen där funktionen kallades på.

Exempel på funktionskall är:

```
(1,2,3)summa ← kallar på funktionen summa i avsnitt om funktionstilldelning  
              returnerar värdet 6
```

2.8 Selektion-satser

Detta avsnitt besvarar vad en selektions-sats innebär, hur de används och hur syntaxen för den ser ut i Domo. En selektions-sats kommer presenteras i dess beståndsdelar.

En selektions-sats används för att kunna välja mellan flera alternativ beroende på ifall huruvida ett uttryck är sant eller falskt. Om uttrycket ger ett sant sanningsvärde kommer blocket tillhörande selektions-satsen att köras, om det ger ett falskt sanningsvärde kommer blocket att hoppas över i körningen av programmet.

Selektions-satser i Domo har som returvärde noll för påvisad lyckad körning.

2.8.1 If-satser

En if-sats används för att kunna ge programmeraren möjligheten att förgrena sin kod. En enkel if-sats är konstruerad i två delar, ett uttryck som utvärderar ifall dess sanningsvärde och ett tillhörande block som körs ifall sanningsvärdet blir sant. Sedan kan programmeraren välja att lägga till en kopplad if-sats som vanligtvis kallas `elsif` som består av samma två delar i sin konstruktion som den enkla if-satsen. Skillnaden är att `elsif`-satsen endast utvärderar sitt uttryck om den kopplade if- eller `elsif`-satsen ovanför blir falsk, om den blir sann kommer resterande `elsif`-satser nedan att hoppas över i körningen av programmet. Det kan läggas godtyckligt många `elsif` satser som är sammankopplade på detta sätt. Till sist kan programmeraren välja att lägga till en sista sats som kommer att köras ifall if-satsen och - om det finns - alla ovanstående `elsif`-satser blir falska. Denna typ av sats brukar normalt kallas för `else`. Observera att för att kunna använda en `elsif`- eller `else`-sats så måste det finnas en tillhörande if-sats bunden till de.

Ett exempel på en if-sats är följande:

```
(4,x)@           ← Ger x värdet 4
(1,x)=?          ← Utvärderar 4 är lika med 1, blir falskt, blocket hoppas över
{
  ("Hamnade i en if-sats")p
}
(2,x)=e?         ← Utvärderar 4 är lika med 2, blir falskt, blocket hoppas över
{
  ("Hamnade i den första elsif-satsen")p
}
(3,x)=e?         ← Utvärderar 4 är lika med 3, blir falskt, blocket hoppas över
{
  ("Hamnade i den andra elsif-satsen")p
}
e>               ← Alla övriga fall, blocket nedan körs
{
  ("Hamnade i else-satsen")p
}
```

2.9 Repetition-satser

Detta avsnitt besvarar vad en repetitions-sats innebär, hur de används och hur syntaxen för dessa ser ut i Dopo. Två typer av repetitions-satser är presenterade.

Repetitions-satser används för att kunna upprepa givna satser inom ett block. Om denna styrstruktur inte existerade hade programmeraren varit tvungen att skriva om samma satser, vilket leder till onödig kodupprepning. Det går alltså snabbare för programmeraren att använda en repetitions-sats och samtidigt ökar läsbarheten i programmet.

Repetition-satser i Dopo har som returvärde noll för påvisad lyckad körning.

2.9.1 While-loopen

En while-loop är den ena av två repetitions-satser som finns konstruerad i Dopo. Den är mest användbar om programmeraren vill upprepa ett givet block beroende på ifall ett uttryck är sant eller falskt. Om programmeraren inte vet på förhand hur länge upprepningen ska pågå är en while-loop att föredra.

Här är ett exempel på en while-loop i Dopo:

```
(1,x)@      ← Ge x värde 1
(x,5)!=x     ← Upprepa nedanstående block tills x = 5
{
  (x)p       ← Skriv ut värdet på x i terminalen
  (1,x)+@    ← Öka värdet på x med x nuvarande värde adderat med 1
}
```

2.9.2 For each-loopen

Den andra repetitions-satsen konstruerad i Dopo är for each-loopen. Denna loop är användbar om programmeraren vill gå igenom en List, String eller Dictionary. Programmeraren vet alltså på förhand hur länge upprepningen ska pågå. Ett vanligt användningsområde för denna repetitions-sats är att hitta ett visst element i en lista eftersom varje element i den givna datatypen som håller på och itereras igenom kommer att kunna finnas tillgänglig i det block som tillhör loopen.

Här är ett exempel på en for each-loop i Dopo:

```
([1,2,3,4,5], lista)@ ← Skapa en List med element 1,2,3,4,5
lista x+ element      ← Först ange vilken variabel som ska itereras igenom,
{                      sedan vad varje element kommer att ha för variabelnamn i blocket
  (element)p
}
```

2.10 Kontrollsatser

I detta avsnitt presenteras funktionalitet för att avbryta repetitions- och selektions-satser samt returnera värden.

2.10.1 Abort

En abort-sats används för att primärt kunna avbryta närmaste nästlade loop i programmet, men den kan också avbryta if-satser som befinner sig utanför loopar. Den kan vara användbar ifall programmeraren vill testa olika delar av programmet genom att kolla hur nuvarande variabeluppsättning ser ut innan abort-satsen. Observera att satserna under abort-satsen inte kommer att exekveras ifall den körs i det globala blocket.

Ett exempel på en abort-sats är följande:

```

([1,2,3,4],lista)@      ← lista får elementen 1,2,3,4
lista x+ element        ← kör en for each-loop
{
  (element,3)=?         ← kör en if-sats
  {
    ("abort körs nedan")p
    !x                  ← abort-satsen körs, hoppar ur if-satsen och loopen
    ("denna sats körs inte")p
  }
}

```

2.10.2 Continue

En continue-sats fungerar likt en abort-sats eftersom den också avbryter if-satser utanför loopar och går vidare till nästa sats samt att programmet avbryts om den körs i det globala blocket. Skillnaden är när en continue-sats exekveras i en loop kommer resten av koden under satsen att ignoreras och vi evaluerar loopens uttryck för nästa steg i loopen. Det gör den användbar för att på ett enkelt sätt kunna kontrollera flödet i en loop.

Ett exempel på en continue-sats är följande:

```

([1,2,3,4],lista)@      ← lista får elementen 1,2,3,4
lista x+ element        ← kör en for each-loop
{
  (element,3)=?         ← kör en if-sats
  {
    ("continue körs nedan")p
    x+                  ← continue-satsen körs, hoppar ur if-satsen och evaluerar loopen igen
    ("denna sats körs inte")p
  }
}

```


2.10.3 Return

Denna sats syftar till att returnera värdet som står framför symbolen och satsen symboliseras med <-. Om denna sats anges i ett funktionsblock kommer satserna nedan att ignoreras och värdet returneras där funktionskallet sker. Om retur-satsen sker utanför en funktion kommer returvärdet att returneras till vi når det globala blocket. Om satsen används i det globala blocket kommer värdet innan retur-satsen att returneras och nedanstående satser att ignoreras.

Om det är tomt framför retur-satsen returneras nil.

2.11 Reserverade namn

Vissa namn är bunda till funktionalitet för språket och kan därav inte användas som namn för användarens funktioner, här är en lista med dem samt en kort beskrivning av vad de gör.

2.11.1 p

Används för utskrift till terminalen, tar parameter och skriver ut dem samt returnerar parametern oförändrad. Tex. ("hello world")p. En tom parameterlista innebär att en blankrad skrivs ut i terminalen.

2.11.2 g

Används för inmatning från terminalen. Väntar på inmatning från terminalen och returnerar sedan det inmatade värdet tolkat som en variabel i språket. T.ex. (indata)g. En tom parameterlista innebär att indata enbart returneras i programmet där kallet görs.

2.12 Stdlib

I stdlib filen finns förskrivna funktioner vilka hanterar listor och strängar, dessa är mer avancerade användningar av operatorerna (se rubrik "Operatorer") och allt som existerar i denna fil läses in och finns tillgängliga som funktioner när språket exekveras.

Dessa funktioner kan dock skrivas över av användaren i egna filer och är på så sätt skilda från "Reserverade namn" se *Tabell 1* för full lista av funktioner i stdlib.

2.12.1 Funktionslista

Tabell 1 Redogör funktioner i stdlib som verkar på sammansatta datatyper i Dopo.

(var)len	Returnerar längden av "var" som ett heltal om den är en lista eller sträng
(list)last	Returnerar sista elementet i "list"
(item,list)push*	Lägger till parametern "item" först i listan "list"
(index, list)del_index*	Tar bort elementet på plats "index" ur listan "list"
(item,list)get_index	Hämtar index på elementet "item" i "list"
(item,list,multiple)del_item*	Tar bara bort första förekomsten av "item" i list om "multiple" är falsk, annars tar bort alla förekomster av "item" i "list"
(string,delim)split	Returnerar en lista med element vilka består av bokstäverna i "string" uppdelade vid symbolen "delim"
(item, list)contains	Returnerar sant om "item" finns i "list" annars returnerar falskt
(list)first	Returnerar första elementet i "list"
(index,list)pop*	Tar bort elementet på plats "index" i "list" och sedan returnerar elementets värde
(item,list)append*	Lägger till parametern "item" sist i listan "list"
(item,index,list)insert*	Lägger till "item" på platsen "index" i "list"

Observera: Funktioner i tabell 1 vilka har en "*" efter sig är funktioner skriva med referensparametrar.

2.13 Operatorer

I språket finns ett antal operatorer vilka utför olika operationer, dessa fungerar olika beroende på datatyp. Operatorerna ändrar inte på parametrarna utan returnerar endast värden. Nedan följer samtliga operatorer samt vad de returnerar för respektive datatyp. Nämns inte datatypen så finns ingen definition för den typen inom den operatorn. Nämns "sant" och "falskt" menas booleska sanningsvärden.

Det går att kedja godtyckligt många operatorer, men parametrarna måste tillhöra samma datatyp för att fungera korrekt.

Observera: Om inte annat nämns utförs envägs operationer så som borttagning eller tilläggning på variabeln längst till vänster.

Observera: Listfunktionalitet appliceras så fort en av parametrarna är en lista.

Observera: Dictionary appliceras endast om Dictionary-objektet är längst till vänster och resterande element är giltiga par

2.13.1 Op: +

String_c: Sammansätt string av alla parametrar

Number_c: Addition summa av parametrar

Bool_c: Logiskt 'or' resultat

Nil_c: nil

List: Sammansatt lista av parametrar i vänster till höger ordning

Dictionary: Dictionary först i parameterlistan med alla giltiga par i parameterlistan tillagda.

2.13.2 Op: -

String_c: Sträng med första förekomsten av en delsträng borttagen

Number_c: Subtraktions differens av parametrar

Nil_c: nil

List: Lista med givet element borttaget

Dictionary: Dictionary med paret med given nyckel borttagen.

2.13.3 Op: *

Number: Multiplikativ produkt av parametrar

Nil: nil

2.13.4 Op: /

Number: Divisions kvot av parametrar, om alla parametrar är heltal utförs heltalsdivision medan om en parameter har decimaler så utförs "vanlig" division med decimaler.

Nil: nil

2.13.5 Op: %

Number: Modulo resultat av vänstra parametrar på högerparameter

Nil: nil

2.13.6 Op: ^

Number: Potensresultat med vänsterparametrar som potenser på högerparameter

Nil: nil

2.13.7 Op: >=

String: sant om värdet till vänster är alfabetiskt längre fram eller på samma plats

tex. ("b","a")>= ger sant

 ("a","a")>= ger sant

 ("a","b")>= ger falskt

Number: sant som vänsterparametern är större eller lika med högervärdet

Nil: nil

2.13.8 Op: <=

String: sant om värdet till höger är alfabetiskt längre fram eller på samma plats

tex. ("b","a") <= ger falskt

 ("a","a") <= ger sant

 ("a","b") <= ger sant

Number: sant som högerparametern är större eller lika med högervärdet

Nil: nil

2.13.9 Op: <

String: sant om värdet till höger är alfabetiskt längre fram

tex. ("b","a") < ger falskt

 ("a","a") < ger falskt

 ("a","b") < ger sant

Number: sant som högerparametern är större än högervärdet

Nil: nil

2.13.10 Op: >

String: sant om värdet till vänster är alfabetiskt längre fram

tex. ("b", "a") > ger sant

("a", "a") > ger falskt

("a", "b") > ger falskt

Number: sant som vänsterparametern är större än högervärdet

Nil: nil

2.13.11 Op: =

String: sant om strängarna består av samma symboler i samma ordning, annars returnerar falskt

Number: sant det är samma nummer i alla parametrar, annars returnerar falskt

Boolean: sant om det är matchande värde i alla parametrar, annars returnerar falskt

Tex. (T,T)= ger sant

(F,F)= ger sant

(T,F)= ger Falskt

Nil: nil

List: sant om samma antal parametrar finns och alla parametrar matchar respektive index i alla andra listor. Annars returnerar falskt.

Dictionary: sant om samma antal par finns och alla nycklar och värden matchar respektive index i alla Dictionary-objekt. Annars returnerar falskt.

2.13.12 Op: !=

String: sant om operatören "!=" returnerar falskt

Number: sant om operatören "!=" returnerar falskt

Boolean: sant om operatören "!=" returnerar falskt

Nil: nil

List: sant om operatören "!=" returnerar falskt

Dictionary: sant om operatören "!=" returnerar falskt

2.13.13 Op: !

Observera: operatören tar endast en parameter

String: falskt

Number: falskt

Boolean: inverterade sanningsvärdet, sant blir falskt och falskt blir sant

Nil: sant

List: sant om listan är tom

Dictionary: sant om Dictionary är tom

2.13.14 Op: |

Boolean: logiskt "or" resultat

Nil: nil

2.13.15 Op: &

Boolean: logiskt 'and' resultat

Nil: nil

2.13.16 Op: ->

Boolean: logiskt 'implikation' resultat, dvs. värde1 kan leda till värde2

Nil: nil

2.14 Metafunktionlitet

2.14.1 Kommentarer

Kommentarer inleds med `"""`-tecken och avslutas med `"""`-tecken. Kan sträckas över flera rader så länge det finns ett start `"""`-tecken och ett slut `"""`-tecken. Allt mellan de två `#` symbolerna kommer då inte tolkas som kod.

3 Systemdokumentation

3.1 BNF-grammatik

Nedan presenteras BNF-grammatiken som beskriver syntaxen för programmeringsspråket. Notationen utgår ifrån *Extended BNF* (EBNF) som tar med konstruktioner för repetition som betecknas med "{}" och selektion "[]". Varje "<>" kommer utgöra en specifik matchgrupp som har en viss funktionalitet i språket. Vidare anges enkla citattecken (") om de är en del av syntaxen för språket. Det är alltså skillnad om det står {} eller '{ }' där den första beskriver en repetition av det som befinner sig inom dessa måsvingar, den andra tydliggör att tecknet måste finnas med i syntaxen.

```
<program> ::= <valid_list>
```

```
<valid> ::= <include>
          | <control>
          | <assignment>
          | <call>
```

```
<include> ::= 'include' <string> { <string> }
```

```
<control> ::= <return>
             | <abort>
             | <continue>
             | <for_each>
             | <while>
             | <if>
```

```
<return> ::= <variable_assignment> '<-'
            | [ <call> ] '<-'
```

```
<abort> ::= '!␣'
```

```
<continue> ::= '␣+'
```

```
<for_each> ::= <call> '␣+' <name> '{' { <valid> } '}'
```

```
<while_loop> ::= <call> '␣' '{' { <valid> } '}'
```

```
<if> ::= <call> '?' '{' { <valid> } '}' { <call> 'e?' '{' { <valid> } '}' } ['e'] '{' { <valid> } '}'
```

```
<assignment> ::= <function_assignment>
                | <variable_assignment>
```

```
<function_assignment> ::=
    | '(' [ <parameter_name> ] { ',' <parameter_name> } ')' '@' <name> '*' '{' { <valid> } '}'
    | '(' [ <parameter_name> ] { ',' <parameter_name> } ')' '@' <name> '{' { <valid> } '}'
```

```

<variable_assignment> ::= '(' <call> ',' <name> ')' <math_op> '@'
                        | '(' <call> ',' <name> ')' '@'
                        | '(' <name> ')' '@'

```

```

<parameter_name> ::= <letter> { <char> }

```

```

<call> ::= <user_input>
        | <print>
        | '(' [ <call> ] { ',' <call> } ')' <name>
        | '(' <call> { ',' <call> } ')' <op>
        | <index>
        | <var>

```

```

<user_input> ::= '(' [ <call> ] ')' 'g'

```

```

<print> ::= '(' [ <call> ] ')' 'p'

```

```

<op> ::= <boolean_op>
        | <math_op>

```

```

<boolean_op> ::= [&, |, !, >=, <=, <, >, =]{1}

```

```

<math_op> ::= [+ , - , * , // , / , ^ , %]{1}

```

```

<index> ::= '[' <call> ']' <call>

```

```

<var> ::= <nil>
        | <dictionary>
        | <list>
        | <string>
        | <bool>
        | <name>
        | <number>

```

```

<nil> ::= 'nil'

```

```

<dictionary> ::= '[' <call> '@' <name> { ',' <call> '@' <name> } ']'
               | '[' '@' ']'

```

```

<list> ::= '[' <call> { ',' <call> } ']'
         | '[' ']'

```

```

<string> ::= '/'[^"]*/

```

```

<number> ::= <float>
           | <int>

```

```

<float> ::= ['-'] {<digit>} '.' {<digit>}

```

```

<int> ::= ['-'] <digit> {<digit>}

```



```

<bool> ::= 'true'
        | 'false'
        | 'T'
        | 'F'

<name> ::= [A-Za-z][\wåäöÅÄÖ]*

<char> ::= <letter>
        | <digits>
        | <symbol>

<letter> ::= [a-zA-Z]

<digits> ::= [0-9]+

<symbol> ::= [$, £, å, ä, ö, Å, Ä, Ö, _]

```

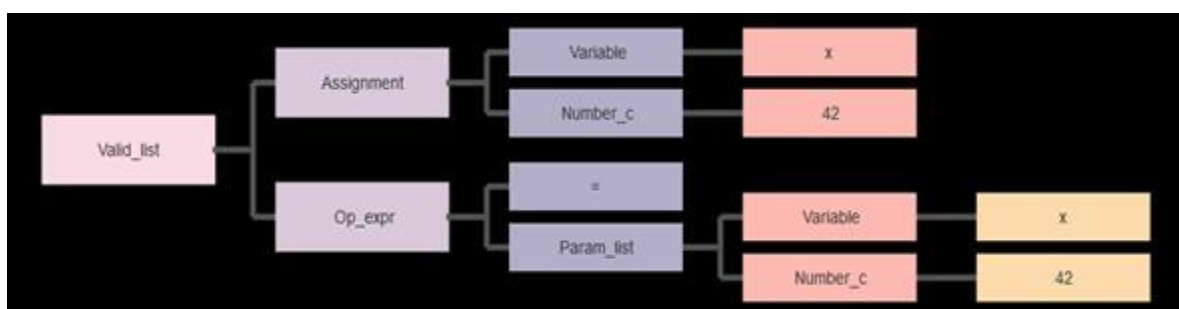
3.2 Lexning/parsning

För lexning samt parsning använder sig språket av en given RDParse parser vilken är en "recursive descent parser". Den används först för att dela upp text i tokens vilka representerar operatorer, uttryck, variabler, text med mera i språket.

För att matcha tokens enligt de grammatiska reglerna (se avsnittet "BNF-grammatik") i språket används parser-delen av RDParse och efter varje slutmatchning skapas ett objekt av en viss klass vilket länkas med andra objekt genom att ett program representeras som en "valid_list". Den innehåller en lista med giltiga uttryck som i sin tur innehåller under-uttryck. På så sätt skapas ett abstrakt syntaxträd.

Objekten är ordnade enligt en arvsstruktur med ett "valid"-statement som grundbyggstenen vilken alla uttryck ärver av, sedan finns ytterligare underklasser för mer specialiserade objekt.

Figur 6 nedan anger ett exempel på ett enkelt syntaxträd.



Figur 6 enkelt syntaxträd uppbyggd av klassnoder

3.3 Exekvering

När parsern har genererat ett syntaxträd bestående av klasser kommer toppen av trädet att starta exekveringen av programmet genom att kalla på funktionen "execute" som är en medlemsfunktion hos alla skapade klasser i språket. Detta kall kommer på så sätt ordna att varje klassobjekt i trädet kommer kalla på sin egen execute-funktion längs syntaxträdets grenar. Resultatet för varje execute-kall kommer sedan att returneras till noden ovanför tills vi når toppen och programmet returnerar ett värde.

3.4 Nod Klasser

Detta avsnitt är till för att programmerare ska kunna ta del av hur en klass är uppbyggd för att vidare kunna underhålla eller utöka språket. För att uppfylla detta redogörs det nedan vad som kännetecknar en typisk klass i Dopo. Även presenteras vissa utvalda klasser som är mest komplicerade och som kan behöva redogöras utförligare.

3.4.1 Allmänt om klasserna

Nästan varje klass i språket utgörs av en "initialize"- och "execute"-medlemsfunktion. Initialize är en konstruktör och syftar till att skapa ett objekt av klassen med dess parametrar som medlemsvariabler. Execute är den funktionalitet som gör klassen unik. Sedan kan det finnas övriga hjälpfunktioner till klasserna för att antingen tydliggöra eller utöka funktionaliteten. Varje klass ärver ifrån Valid-klassen.

Utöver det globala scopet kommer det skapas nya scopes när ett kall på en funktion görs i klassen Call, när det skapas en loop eller när det skapas en if-sats. Efter varje tillhörande block har utfört sin exekvering kommer varenda en av dessa klasser att ta bort det skapade scope med dess innehåll av variabler och funktioner. Enbart det globala scopet existerar under hela förloppet av programmet.

3.4.2 Dopo

Denna klass finns i filen "Dopo.rb" och innehåller hela lexer- och parser-delen. Denna klass har även som klassvariabel en array av samtliga scopes i programmet och använder har klass-funktioner för att kunna lägga till och ta bort scopes samt att kunna lägga till individuella variabler eller funktioner i scopet. Varje scope och enskild funktion är i sin tur en egen klass med egna medlemsfunktioner för hantering av dess medlemsvariabler.

3.4.3 Valid_list

Denna klass har samtliga noder i syntaxträdet som existerar i programmet representerade i Valid- och Valid_list-objekt som medlemsvariabler. Syftet med klassen är att exekvera programmet samt att göra typcheckar ifall ett Valid-objekt returnerar ett Abort-, Continue eller Return_statement-objekt. Denna funktionalitet är nödvändig för att manipulera vad vi returnerar och på så sätt kan vi simulera hur dessa klasser ska fungera i praktiken. Detta genomförs genom att kalla på hjälpfunktioner och sätta vissa klassvariabler beroende på utfall.

Den generella regeln är att ifall det sparade valid-objektet returnerar ett Abort- eller Continue-objekt ska nuvarande Valid_list-objekt fortsätta returnera samma objekt till vi hamnar till närmast nästlade loop.

När exekveringen returnerar objekt av Return_statement sparas värdet som returneras undan i en klassvariabel och vi fortsätter att returnera samma objekt likt stycket ovan. Om exekveringen befinner sig i en loop eller if-sats returneras ett Abort-objekt för att hoppa ur dessa styrsatser. Denna process utförs tills vi når platsen för kallet där värdet för return-satsen ska placeras.

3.4.4 Call

Syftet med denna klass är att utföra ett kall på en funktion. Detta innebär att exekvera varje parameter i parameterlistan. Hitta funktionen utifrån ett funktionsnamn. Gör en kontroll ifall funktionen är så kallad "öppen", om så är fallet då ska varje parameter som skickas in få värdet som parametrarna i funktionsdeklarationen får i slutet av exekveringen av funktionsblocket.

Denna klass har som medlemsvariabler namnet på funktionen som ska kallas på och parameterlistan som förs med i kallet.

Denna klass utför kontroller för att se till att funktionen som kallas på existerar i ett tillgängligt scope och att rätt antal parametrar anges vid kallet.

3.4.5 Operator_expr

Denna klass hanterar funktionalitet för operator-funktioner för alla datatyper vilka operatorer fungerar på. Klassens *execute* har som uppgift att hitta vilken data typ som finns i parametrarna och därefter kalla på rätt hjälpfunktion vilken opererar på den sortens data typ. Dessa hjälpfunktioner kontrollerar att en giltig operator används samt utför operationen och returnerar ett nytt värde. Denna klass kastar även fel när en ogiltig operator eller ogiltiga kombinationer av parametrar används. Rubys *eval* funktionalitet används i vissa fall där det är smidigt, utöver det skrivs funktionalitet under en hittad operator.

3.5 Algoritmer

Språket körs genom Ruby och använder sig Rubys inbyggda algoritmer för exempelvis matematiska operationer, list funktioner, minneshantering med mera.

3.6 Kodstandard

En kodstandard representerar som en mall på hur man ska skriva kod, nedan följer riktlinjer kring hur koden som bygger upp språket är skriven då ingen existerande kodstandard används.

3.6.1 Indentering

Dubbelt blanksteg används för indentering då det är tydligare att läsa än singulärt blanksteg och kompaktare än fler blanksteg.

3.6.2 Kommentarer

Kommentarer skrivs för varje klass som förklarande text samt för varje icke-trivial funktion och svårläst kod som förklaring till kod delens syfte.

3.6.3 Klassnamn

Klasser skrivs med stor bokstav samt understreck mellan nya ord.

till exempel:

- `class Else_if_list < Valid`

3.6.4 Variabelnamn

Variabler skrivs med små bokstäver med nya ord separerade med understreck, medlemsvariabler inleds med @.

Till exempel:

- `statement`
- `@statement`
- `else_statement`
- `@else_statement`

3.6.5 Funktionsnamn

Funktionsnamn skrivs med små bokstäver med nya ord separerade med understreck.

- till exempel:
- `execute_dict_func`
- `execute`

4 Erfarenheter och reflektion

I början av projektet var implementationsplanen som skrevs användbar, då den följdes och gav riktning till projektet vid varje arbetspass. Desto mer av projektet som blev implementerat desto fler saker kunde tolkas på olika sätt. Och i vissa fall blev det helt annorlunda gentemot implementationsplanen. Detta var dock inte en nackdel på grund av att de olika delarna var modulära och inte förstördes på grund av ändringar i andra delar. Förberedelserna räckte bra till för att implementera stora delar av projektet därav anses BNF-grammatiken och implementationsplanen vara kompletta redan vid projektstart. Utifrån detta visas att förberedelser sparar tid och är värdefullt under hela projektets gång.

På grund av brist på erfarenheter inom liknande projekt sattes milstolpar pessimistiskt och arbetstakten var hög. Detta ledde till många milstolpar blev uppnådda långt före deadline. Projektmedlemmarna har nu en bättre tidsuppfattning kring projekt av denna typ.

Flera gånger under projektets gång ansågs milstolpar vara klara för att sedan efter ytterligare implementationer insågs de vara ofullständiga. Detta var till stor del ett resultat av icke kompletta tester, vilka testade grundläggande funktionalitet men inte komplett funktionalitet. Efter upprepande misstag av denna typ skrevs ett bättre test-system med utförliga tester och nya funktioner fick dedikerade tester. Lärdomen av detta är att tester är hjälpsamma och bör utföras grundligt från start i ett projekt samt uppdateras vid tillägg av ny funktionalitet.

Allt i implementationsplanen implementerades. Utöver implementationsplanen implementerades även förkortade funktioner av operatorresultat med tilldelning till en variabel ($+=$, $-=$, etc.), importering av filer via nyckelordet *include*, datatypen Dictionary, for each-loop samt användarinmatningsfunktion från terminalen. På grund av dessa inte var med i implementationsplanen var de inte lika självklara i implementation, då andra delar hade diskuterats i förhand medan dessa inte var förberedda. Implementationsplanen kunde ha varit mer omfattande och inkluderat fler funktionaliteter.

I början användes parprogrammering för att bygga en grundfunktionalitet för språket. Sedan varierades arbetssättet också med uppdelning av implementationer. Löpande under projektet användes versionshantering via git. Vid uppdelning av implementationer sattes tydliga och separerade uppgifter och mål. Detta krävde att delarna blev modulärt skrivna, vilket var en grundprincip. Parprogrammering är effektivt på grund av att kunna få kod direkt granskad och kommenterad, vilket minskar antalet fel och ger bra förståelse för samtliga involverade. Ett resultat av detta är att de involverade kan få förståelse och lättare kan utöka koden med egna delar. Till skillnad ifrån om endast en person hade skrivit koden, vilket hade krävt att resterande hade behövt först läsa och förstå koden. En annan lärdom är att versionshantering underlättar utveckling eftersom ändringar lätt kan återställas och tidigare versioner av kod är lättåtkomliga. Frekventa och välformulerade commits underlättar läsbarheten av ändringar och blir viktigare desto fler medlemmar som är involverade i projektet.