LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Cobra Dokumentation

Albin Dahlén, albda746@student.liu.se
Filip Ingvarsson, filin764@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2022
Version 1.1

2023-05-31

Innehåll

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första utkast	23-05-09
1.1	Korrigering efter feedback	23-05-31

1 Inledning

Det här projektet genomfördes som en del av programmet Innovativ Programmering vid Linköpings universitet i kursen TDP019: Konstruktion av datorspråk.

Språket som utvecklats kallas för Cobra och har inspirerats mycket av C++ och Python. Cobra tar de hårt typade delarna från C++ men tar också inspiration av den enkla syntaxen från Python genom att skala bort onödiga parenteser runt olika typer av satser. Detta resulterar i ett språk där användaren har stor kontroll över vilka variabler som lagrar vilka värden men slipper det eviga sökandet efter de typiska problemen som kan uppstå i C++, exempelvis som vilken rad semikolonet har missats på. Cobra klassas som ett *general purpose language* och innehåller de flesta funktioner som ett sådant språk kräver.

1.1 Målgrupp

Målgruppen för detta projektet är programmerare som är intresserade av ett typat språk utan onödiga parenteser och semikolon.

2 Användarhandledning

Denna handledning är gjord för att ge en förståelse i hur Cobra skrivs i programkod samt vilka konstruktioner som existerar i språket och exempel för varje konstruktion.

2.1 Installation

Installationen av språket sker via terminalen och görs med från följande Git. Giten innehåller en README-fil som specificerar vilka kommandon som ska köras för att installera Cobra. Här kommer en kortfattad steg för steg-guide hur språket installeras och körs.

1. Se till att ruby och git är installerade
2. Hämta ner filerna från giten på valfritt sätt och placera i en katalog på datorn.
3. Skapa en text-fil med ändelsen cobra (fil.cobra)
4. I filen som skapades i föregående steg skriv koden för ditt program enligt syntaxen (Se avsnittet Konstruktioner)
5. Kör sedan programmet med koden:

```
ruby main.rb fil.cobra
```

Listing 1: Kommando för att köra språket på en fil

2.2 Konstruktioner

Språket innehåller flera konstruktioner som beskrivs nedan. Beskrivningarna är relativt grundläggande. En mer detaljerad beskrivning ges i den autogenererade-dokumentationen.

2.3 Datatyper

Datatyper, eller ofta kallat variabler, är ett sätt att spara information inom programmering. En datatyp kan liknas vid en låda där informations kan stoppas undan för att användas senare. Variabler skapas med ett namn. I Cobra måste dessa namn börja med en bokstav (a-z) följt av bokstäver (a-z) siffror(0-9) eller understreck (_). Namnen är inte skiftlägeskänsliga vilket betyder att det inte finns någon skillnad på versaler och gemener.

Alla datatyper har ett antal metoder. De metoder som finns definierade för alla datatyper är:

- **copy()**: Returnerar en kopia av objektet.
- **get_type()**: Returnerar en sträng med objektets typ.
- **to_s()**: Returnerar objektet i strängformat.

2.3.1 Int

Variabler som innehåller heltal kallas för int. En int består av ett positivt eller negativt heltal. Datatypen representeras av klassen *NumberVal*.

```
int a
a = 5
int b = a + 3
```

Listing 2: Int-exempel

I listing 2 visas först hur en integer vid namn a skapas. Därefter tilldelas den värdet 5. Efter detta tilldelas en integer vid namn b med värdet av a+3, alltså värdet 8.

Metoder

Ingen av metoderna som finns i *NumberVal* modifierar objektet utan returnerar en kopia istället. Nedan finns de metoder som är tillgängliga för instanster av typen *NumberVal*. Exempel för dessa metoder finns i dokumentationen som går att generera med kommandot *rake doc*.

- **to_int()**: Konverterar det nuvarande talet till ett heltal.
- **to_float()**: Konverterar det nuvarande talet till ett decimaltal.
- **round(num)**: Avrundar talet till *num* decimaler.

2.3.2 Float

Variabler som innehåller decimaltal kallas för float. En float består av ett negativ eller positiv decimaltal. Datatypen representeras av klassen *NumberVal*.

```
float a = 3.7
a = a - 1.0
```

Listing 3: Float-exempel

I listing 3 visas hur en float variabel skapas i Cobra. En float med namnet a skapas och tilldelas värdet 3,7. Därefter tilldelar vi a med värdet av a minus 1,0. A får då värdet 2,7.

Metoder

Ingen av metoderna som finns i *NumberVal* modifierar objektet utan returnerar en kopia istället. Nedan finns de metoder som är tillgängliga för instanster av typen *NumberVal*. Exempel för dessa metoder finns i dokumentationen som går att generera med kommandot *rake doc*.

- **to_int()**: Konverterar det nuvarande talet till ett heltal.
- **to_float()**: Konverterar det nuvarande talet till ett decimaltal.
- **round(num)**: Avrundar talet till *num* decimaler.

2.3.3 String

Variabler som innehåller tecken kallas för string. En string, eller svenskans sträng, kan innehålla tecken, likt vanlig text. För att skilja på programtext och texter som ska tolkas som strängar omfamnas strängar med dubbel-citattecken(") eller enkel-citattecken('). Det viktiga är att en sträng måste använda samma typ av citattecken både i början och i slutet för att tolkas som en giltig sträng. Datatypen representeras av klassen *StringVal*.

```
string x = "Det här är en sträng i Cobra"
string y = 'Även detta är en sträng i Cobra'
```

Listing 4: String-exempel

I listing 4 visas hur strängar kan skapas i Cobra. Det fungerar både med dubbel-citattecken(") eller med enkel-citattecken('). För en sträng måste båda dessa tecken vara samma.

Metoder

Ingen av metoderna som finns i *StringVal* modifierar objektet utan returnerar en kopia istället. Nedan finns de metoder som är tillgängliga för instanster av typen *StringVal*. Exempel för dessa metoder finns i dokumentationen som går att generera med kommandot *rake doc*.

- **length()**: Returnerar längden på strängen.
- **to_array()**: Konverterar strängen till en array där varje bokstav är ett värde i arrayen.
- **to_int()**: Konverterar strängen till ett heltal.
- **to_float()**: Konverterar strängen till ett decimaltal.

2.3.4 Bool

Bool är en datatyp som innehåller ett sanningsvärde "true" eller "false". Datatypen representeras av klassen *BooleanVal*. En bool har inga metoder utöver de som specificerats i avsnitt 2.3.

```
bool a = true
bool b = false
```

Listing 5: Boolean-exempel

I listing 5 visas hur booleans skapas i Cobra. En boolean a skapas med värdet "true" och en boolean b skapas med värdet "false".

2.3.5 Array

Om andra datatyper kunde liknas vid en låda kan en array liknas vid en samling av lådor som innehåller ett värde. Arrayer är indexerade där första värdet är "plats" 0 i arrayen. Arrayer har en typ som berättar vilken typ av värden som är tillåtna att lägga till i arrayen. Det är därför inte möjligt att ha en array som tar både värden av typen int och typen string. Datatypen representeras av klassen *ArrayVal*.

```
int[] a = int[]{1,2,3,5}
string[] x = string[]{"första", "andra", "sista"}
```

Listing 6: Array-exempel

I listing 6 visas exempel på hur arrayer kan skapas i Cobra. Den första arrayen är av typen int som innehåller heltalen 1-5. Den andra array av typen string som innnehåller strängarna "första", "andra", "sista".

För att komma åt värden på en specifik plats inuti arrayer finns det en speciell funktion se listing 7.

```
int[] a = int[]{1,2,3,4,5}
int b = a[3]
```

Listing 7: Åtkomst till värden i en array

I listing 7 skapas en array med heltalen 1-5. Sedan tilldelas variabeln b talet som finns på indexplats 3. Något som är viktigt att komma ihåg är att arrayer är 0-indexerade, alltså att de börjar räkna på noll. Detta betyder att värdet på index 3 är det 4:e elementet i arrayen. Variabeln b kommer alltså få värdet 4.

Vid modifiering av innehållet på en specifik indexplats kan ovanstående funktion användas men med en liten ändring.

```
int[] a = int[]{1,2,3,4,5}
a[2] = 6
```

Listing 8: Ändra värden i en array

I listing 8 ändras värdet på indexplats 2 i arrayen, alltså heltalet 3 i detta fallet, till heltalet 6.

```
int[][] a = int[][]{int[]{1,2,3,5}, int[]{10, 2}}
int b = a[1][0]
```

Listing 9: Nästlad-array

Arrayer kan även vara nästlade till ett godtyckligt djup. I listing 9 finns ett exempel på ett tvådimensionell-array, en array av arrayer. För att komma åt ett element i en nästlad-array behöver man chaina access-operatorn. I lisiting 9 så kommer variabeln b att får värdet 10. Den första access-operatorn säger att array på index 1 ska hämtas sedan ska värdet på index 0 hämtas från den arrayen.

Metoder

Nedan finns de metoder som är tillgängliga för instanster av typen *ArrayVal*. Exempel för dessa metoder finns i dokumentationen som går att generera med kommandot *rake doc*.

- **length()**: Returnerar antalet element i arrayen.
- **pop()**: Tar bort det sista elementet i arrayen och returnerar det.
- **append(append_value)**: Lägger till ett element sist i arrayen.
- **remove_at(index)**: Tar bort ett element på ett specifikt index och returnerar det.
- **sort(function)**: Sorterar elementen i listan enligt en använd-definierad-funktion.

2.3.6 Hash

En hash eller hashtabell är likt arrayer en datatyp som lagrar flera värden. Här är dock värdena inte indexerade utan de lagras med en unik nyckel. Varje nyckel måste vara unik och kan bara hålla ett värde. Datatypen representeras av klassen *HashVal*.

```
Hash<string, string> my_hash = Hash<string, string>{ 'linköping'='LITH', 'stockholm'='KTH' }
string x = my_hash['linköping']
```

Listing 10: Hash-exempel

Listing 10 är ett exempel på en hash i Cobra. I detta exemplet skapas en hashtabell med nycklar av string-typ och värden av string-typ. Tilldelar också värdet ”LITH” till nyckeln ”linköping”, och värdet ”KTH” till nyckeln ”stockholm”. Tilldelar värdet av nyckeln ”linköping” från hashen my_hash till strängen ”x”, alltså värdet ”LITH”.

Metoder

Nedan finns de metoder som är tillgängliga för instanster av typen *HashVal*. Exempel för dessa metoder finns i dokumentationen som går att generera med kommandot *rake doc*.

- **length()**: Returnerar antalet element i hashen.
- **keys()**: Returnerar en array med alla nycklar som finns i hashen.
- **values()**: Returnerar en array med alla värden som är sparade i hashen.
- **clear()**: Tömmer alla värden i hashen.

2.3.7 Null

Null är en datatyp i språket som används för att representera när ett värde inte finns. Datatypen representeras av klassen *NullVal* och har inga metoder.

2.4 Tilldelning av variabler

I ovanstående exempel har den enklaste typen av tilldelning används, nämligen ”=”, men i Cobra finns det flera typer av tilldelning.

Dessa är ”+=”,”-=”,”*=”,”/=”. De kan ses som att operationen som står på den vänstra sidan av likhetstecknet utförs på variabeln och det som tilldelas. Alla dessa operationer är utbytbara med att skriva ”variabeln = variabeln operatorn värdet”.

```
int x = 10
x += 10
x -= 5
x *= 2
x /= 3
```

Listing 11: Tilldelningsexempel

Variabeln x kommer efter varje steg i listing 11 vara: 10,20,15,30,10

2.5 Överföring av variabler

Tilldelning av variabler som är primitiver, det vill säga int, float, bool och string, ger värdet till variabeln. Komplexa datatyper ger istället en referens vid tilldelning om inte funktionen ”.copy()” används på objektet se listing 12. En referens betyder att de två variablerna innehåller exakt samma objekt.

```
int[] array1 = int[1, 2, 3]
int[] array2 = array1.copy()
int[] array3 = array1
```

Listing 12: Referens eller kopia-exempel

Detta skulle resultera i att array2 skulle bli en kopia av array1, medan array3 är en referens till array1. Är variabel en referens betyder det att om ett element läggs till i array3 kommer detta element även att finnas i array1 och vice versa. Då array2 är en kopia så kommer den inte ändras om något ändras i array1 eller array3. Istället kommer array2 att behålla det ursprungliga värdet, i det här fallet [1,2,3].

2.6 Operationer

För att vara ett fullt fungerande språk behöver olika typer av operation kunna utföras med datatyperna. Exempel på detta skulle kunna vara att utföra beräkningar eller jämförelser på olika sätt. Operationer kan också delas in i Aritmetiska operationer, Logiska operationer samt jämförelseoperationer.

2.6.1 Aritmetiska operationer

Aritmetiska operationer är de vanligaste matematiska operationerna, +, -, *, / samt %. Prioriteringsregler för matematiska uttryck i Cobra fungerar likadant som i vanligt matte, vilket betyder att multiplikation, division och modolu-operationer utförs före addition och subtraktion. Det går även att omge ett uttryck med ”()” för att ändra prioriteten. Det som står inuti parentesen kommer att utvärderas före det som står utanför.

```
int a = 3+7-2
int b = 3*10/5
int c = 24%5
int d = 3+(7-3)*12/3
```

Listing 13: Aritmetiska operationer-exempel

I listing 13 visas exempel på olika uträkningar. Värdet variablerna blir tilldelade efter uträkningen är utförd är: int a = 8, int b = 6, int c = 4, int d = 19.

2.6.2 Logiska operationer

Logiska operationer används på booliska värden eller uttryck och ger ett booleskt svar, sant eller falskt. I Cobra betyder && den logiska operationen och, || betyder logiskt eller och ! betyder logiskt inte.

```
bool a = true && false
bool b = true || false
bool c = a || b and a
bool d = !c
```

Listing 14: Logiska operationer-exempel

I listing 14 visas exempel för de olika logiska operationerna i Cobra. Värdet variablerna blir tilldelade efter evalueringen är följande: bool a får värdet false, bool b får värdet true, bool c får värdet true, bool d får värdet false då det är motsatsen till c som har värdet true.

2.6.3 Jämförelseoperationer

Jämförelseoperationer används för att jämföra olika datatyper och ger ett booleskt svar. I Cobra finns operationerna < och > som betyder mindre och större än, <= och >= som betyder mindre eller lika med samt större eller lika, != som betyder inte lika och == som betyder lika.

```
bool a = 3 > 7
bool b = 3 < 7
bool c = 3 <= 3
bool d =  "hej" != "hejdå"
bool e = true == d
```

Listing 15: Logiska operationer-exempel

I listing 15 visas exempel för de olika jämförelseoperationerna. bool a får värdet false då 3 inte är större än 7, bool b får värdet sant då 3 är mindre än 7, bool c får värdet true då 3 är lika eller mindre än 3, bool d får värdet true då ”hej” inte är samma som ”hejdå”, bool e får värdet true då variabeln d har värdet true och är lika med true.

2.7 Villkorssatser

Villkorssatser används för att styra vad som ska hända eller inte hända beroende på om sanningsvärdet är sant eller falskt.

I Cobra finns det tre olika villkorssatser, if, elsif och else. If och elsif har ett vilkor efter sig och om vilkoret är sant kommer koden som står inom blocket efter villkoret att köras. Ett block i Cobra börjar med ”{” och avslutas med ”}”. Skulle If-satsen ha en else-sats efter sig kommer else-satsen att köras om villkoret för if-satsen är falskt. Utan en else-sats kommer inget att köras.

En if-sats kan även följas av flera elsif-satser. En elsif-sats evalueras endast om villkoret för den övre villkorsatsen är falskt. En if-sats kan ha flera elsif-satser men bara en else-sats.

```
int a = 2
string x
if a!=1 {
    x = "första"
    }
else{
    x = "andra"
}
```

Listing 16: Villkorssatser-exempel

I listing 16 kommer x få värdet ”första” då a har värdet 2 vilket inte är samma som 1. Detta gör att villkoret blir sant.

```
int a = 7
string x
if a==1 {
    x = "första"
    }
elsif a ==3{
    x = "andra"
}
elsif a == 5{
    x = "något"
}
else{
    x = "annat"
}
```

Listing 17: Villkorssatser-exempel 2

I listing 17 kommer x få värdet ”annat”.

```
int a = 5
string x
if a==1 {
    x = "första"
    }
elsif a ==3{
    x = "andra"
}
elsif a == 5{
    x = "något"
}
else{
    x = "annat"
}
```

Listing 18: Villkorssatser-exempel 3

I listing 18 är a 5 vilket betyder att första villkoret är falskt. Eftersom if-satsens villkor var falskt kommer villkoret i första elsif-satsen att jämföras. Resultatet av första elsif-satsen kommer också att vara falskt och nästa villkor ska jämföras. I detta fallet kommer villkoret att vara sant och x kommer få värdet ”något”.

2.8 Repetitionssatser

Repetitionssatser används när man vill upprepa en bit i koden utan att behöva skriva samma sak flera gånger. I Cobra finns det tre olika typer av repetitionssatser while-loop, for-loop, och loop över containers.

2.8.1 While-loop

While-loopen består av ett sanningsuttryck och ett block där blocket kommer köras tills sanningsuttrycket är falskt. Efter varje körning av blocket kommer sanninsuttrycket att jämföras igenom och blocket kommer antingen köras minst en gång till eller så är loopen klar.

```
int i = 0
while i < 10 {
    i = i+1
}
```

Listing 19: While-loop-exempel

I listing 19 kommer kodblocket att köras tio gånger och för varje gång kommer ”i” öka med ett.

2.8.2 For-loop

En for-loop kan användas som en while-loop för att upprepa kod flera gånger men är enklare att använda om man vill göra ett visst antal gånger som man vet på förhand. For-loopen börjar med att deklarera en variabel som oftast används som en iterations-variabel. Efter detta följer ett sanningsuttryck som är villkoret för att loopen ska köras, sedan kommer ett uttryck som oftast används för att öka iterations-variabel, detta uttryck körs efter varje iteration. Slutligen följer ett block av kod som ska köras vid varje iteration.

```
int x = 1
for int i = 0, i < 10, i += 1 {
    x = x * 10
}
```

Listing 20: For-loop-exempel

I listing 20 kommer ”x” multipliceras med 10 i varje loop, och blir tillslut 10000000000.

2.8.3 For-loop över containers

I Cobra finns också en inbyggt loop över containers. Detta är användbart om man t.ex vill loopa över alla värden i en array. Loopen skrivs genom att använda nyckelordet ”for” följt av en identifierare som kommer vara namnet inuti kodblocket för det nuvarande värdet i arrayen vi loopar över. Därefter följer ordet ”in” och sedan namnet på containern som ska itereras över.

```
int total = 0
int[] x = int[]{1,2,3,4,5}
for value in x{
    total = total + value
}
```

Listing 21: For-loop-container-exempel

I listing 21 kommer varje värde att adderas till total så totals värde kommer tillslut bli 15.

2.9 Funktioner

Funktioner är ett sätt att skriva en del av sin kod som man vet man kommer upprepa flera gånger. Man kan säga att man återanvänder sin kod utan att behöva skriva den igen. Funktioner fungerar så att de tar godtyckligt många parameterar, vilket kan vara inga alls, och ger tillbaka ett returvärde. Funktioner måste först deklareras och sedan kan de användas. Funktioner i Cobra deklareras såhär:

```
func returtyp funktionsnamn (int parameter1,bool parameter2){
    #kodblock
}
```

Listing 22: Funktions-exempel

Först skriver man nyckelordet ”func” följt av returtypen ens funktion och därefter namnet för funktionen man vill skapa. Därefter ska eventuella parametrar läggas till. Parametrarna skriv inuti parenteserna. En parameter är en variabel som kommer skickas med från huvudprogrammet till funktionen. Man kan skapa funktioner utan några parametrar eller godtyckligt många. Parametrarna är en datatyp följt av dess namn som används inuti funktionen. Parametrar separeras med ”,”. Efter parameterlistan kommer kodblocket som är det som kommer köras när man kallar på funktionen. Här är ett exempel på en funktion:

```
func int doubel_or_remove_one (int value,bool double){
    if double == true{
        return value*2
    }
    else{
        return value-1
    }
}
```

Listing 23: Funktions-exempel 2

I listing 23 visas ett exempel på en funktion. Funktionen ”doubel_or_remove_one” tar två parameterar, ett heltal ”value” och ett sanningsvärde ”double”. Om double är sant så returneras värdet av value multiplicerat med 2, annars returneras värdet av value minus ett.

För att kalla på en funktion i Cobra skrivs:

```
int x = double_or_remove_one(10,true)
int y = double_or_remove_one(10,false)
```

Listing 24: Funktions-exempel 3

I listing 24 kommer x att få värdet 20 och y kommer få värdet 9.

I Cobra finns det även funktioner som inte har något returvärde. Dessa använder nyckelordet ”void” istället för namnet på typen som ska returneras. Skulle funktionen innehålla return kommer evalueringen av funktionen att avslutas tidigare men inget värde kommer returneras. Alltså i en *void-funktion* fungerar return som ett stopp för att forsätta evaluera den funktionen.

```
func void print_values(int[] values) {
    for val in values {
        print(val)
        if val > 10 {
            return
        }
    }
}
```

Listing 25: Void-funktion-exempel

I listing 25 finns ett exempel på en void-funktion som skriver ut alla värden som är mindre än 10 i en array tills ett värde som är större än 10 dyker upp. När ett värde som är större än 10 kommer de resterande värdena inte att skrivas ut.

2.9.1 Inbyggda funktioner

Det finns även inbyggda funktioner i Cobra. Detta är funktioner som går att anropa varsomhelst i programmet. En av dess är ”print()”. Den skriver helt enkelt ut det som finns inuti parameterlistan i terminalen. Den kan ta variabler, värden, eller returvärden från funktioner.

```
print(2)
string x = "hej"
print(x)
print(double_or_remove_one(5,false))
```

Listing 26: Print-exempel

I listing 26 visas hur print används för att skriva ut olika saker till terminalen. På den första raden kommer talet 2 att skrivas ut. Nästa rad deklareras en sträng med värdet ”hej” denna variabel skrivs sedan ut och i terminalen kommer då ”hej” att visas. Den sista raden skrivs resultatet av funktionen *double_or_remove_one* ut.

En annan funktion är ”input()”. Denna funktionen används för att hämta användarinmatad data från terminalen.

```
string x = input()
>> hej
```

Listing 27: Input-exempel

I listing 27 används input-funktionen för att läsa in ett värde från terminalen och sedan tilldela detta till variabeln x. Den andra raden visar hur det ser ut i terminalen när användaren matar in värdet.

2.10 Klasser

Klasser är en konstruktion där man kan samla relaterade variabler och funktioner. Ett exempel skulle vara en bil-klass där man kan spara hur många säten bilen har, vilken färg bilen har och hur snabbt den kan köra. Klasser har en ”constructor” vilket är en funktion som anropas varje gång en klassinstans skapas. Konstruktorn kan ta olika parametrar eller inga alls. I bil-exemplet (se listing 28) används antalet säten, högsta hastigheten samt färgen på bilen. En klass kan även ha flera olika konstruktorer så länge de inte innehåller lika många parametrar av samma typ.

```
class Car{
    int seats
    int maxspeed
    string color
    Constructor(int number_of_seats, int max_speed, string color_of_car){
        seats = number_of_seats
        maxspeed = max_speed
        color = color_of_car
    }
    func get_seats(){
        return seats
    }
    func get_maxspeed(){
        return maxspeed
    }
    func get_color(){
        return color
    }
}
```

Listing 28: Klass-deklaration-exempel

Listing 28 är ett exempel på en bil-klass med en konstruktor. Vid skapandet av en instans behöver antal säten, topphastigheten och vilken färg bilen har skickas med till konstruktorn.

```
class Car{
    int seats
    int maxspeed
    string color
    bool electric
    Constructor(int number_of_seats, int max_speed, string color_of_car){
        seats = number_of_seats
        maxspeed = max_speed
        color = color_of_car
        electric = false
    }

    Constructor(int number_of_seats, int max_speed, string color_of_car, bool is_electric){
        seats = number_of_seats
        maxspeed = max_speed
        color = color_of_car
        electric = is_electric
    }
    func int get_seats(){
        return seats
    }
    func int get_maxspeed(){
        return maxspeed
    }
    func string get_color(){
        return color
    }
    func bool is_electric(){
        return electric
    }
}
```

Listing 29: Klass-deklaration-exempel med flera konstruktorer

Listing 29 är en utökning av exemplet i listing 28. Detta exemplet visar att en klass kan innehålla flera konstruktorer enda kravet är att de har olika parametrar. Detta exemplet utökar Car-klassen med ytterligare en konstruktor som utöver de vanliga sakerna tar ett sanningsvärde ”is_electric”.

För att sedan skapa instanser av sina klasser i Cobra skrivs:

```
new Classname()
```

Listing 30: Klass-instans-exempel

Ytterligare ett exempel från bil-klassen som nämndes tidigare

```
Car ferrari = new Car(2,200, 'red')
Car volvo = new Car(5,100, 'white')
Car tesla = new Car(5, 150, 'white', true)
print(volvo.get_seats())
print(ferrari.get_color())
```

Listing 31: Klass-instans-exempel 2

I listing 31 skapas en bil ferrari och volvo genom att använda den första konstruktorn som tog 3 parametrar. Bilen tesla skapas sedan med den andra konstruktorn som även tog om bilen är elektrisk som ett sanningsvärde.

Utskriften till terminalen skulle se ut enligt följande. »5 »red

2.11 Scope

Ett koncept inom programmering kallas räckvidd eller scope på engelska. Scope finns för att hantera var i programmet en viss variabel existerar. En variabel som skapas inom en if-sats, for-loop eller funktion existerar bara inom det kodblocket och skulle man försöka använda den utanför skulle man få något felmeddelande om att variabel inte existerar. I Cobra har variabler synlighet ”neråt” i scopet, exempelvis en variabel som skapats innan en if-sats kommer synas inuti if-satsen. Variabler dör även med sitt scope vilket betyder att skapas en variabel inuti ett block, exempelvis en if-sats, kommer denna inte att vara tillgänglig utanför detta blocket.

```
int a = 10
if true {
    a = 20
}
print(a)
```

Listing 32: Scope-exempel

I listing 32 deklareras variabeln a med värdet 10 innan if-satsen. I if-satsen får variablen värdet 20. Efter som variabeln är deklarerad utanför if-satsen kommer a nu att ha värdet 20 när print-funktionen anropas.

```
if true {
    int a = 20
}
print(a)
```

Listing 33: Scope-exempel 2

I listing 33 skapas variabeln a inuti if-satsen när blocket sedan avslutas med ”}” kommer variabeln att försvinna. Detta betyder att ett fel kommer uppstå och felmeddelandet variabeln a finns inte deklarerad kommer visas. Detta är pågrund av att variabeln a inte existerar utanför sitt scope, som är inuti if-satsen.

2.12 Kommentarer

En kommentar kan skrivas för att kunna skriva en bit text inuti kodfilen utan att den ska tolkas eller köras i programmet. Detta kan vara användbart om man vill förklara för andra programmerare hur kod fungerar. Kommentarer i Cobra skrivs genom att skriva ”#” och därefter skrivs kommentaren.

```
#this is a comment
```

Listing 34: Kommentar-exempel

3 Systemdokumentation

Systemdokumentationen behandlar hur språket är uppbyggt och hur det fungerar. Delar som lexer, parser och ast-noder kommer gås igenom. Genomgången kommer vara mer översiktligt och inte gå in i detalj på hur alla klasser och metoder fungerar. För en mer djupgående inblick i hur systemet är uppbyggt går det att generera dokumentation för hela systemet. En beskrivning för hur dokumentationen genereras finns i README-filen på Gitlaben.

3.1 Översikt

Språket är byggt ovanpå Ruby och använder Rubys inbyggda klasser och funktioner med mera för att bygga upp språket. All kod läses och parsas av en egenutvecklad parser som består av tre huvuddelar, Lexern, Parsern och Interpretatorn, varje del är ansvarig för en del av evaluaringen av ett program.

Lexern hanterar den lexikaliska analysen, här översätter Lexern användarens kod som är en lång sträng till en lista av tokens. Denna lista av tokens skickas vidare till Parsern som är nästa steg i evalueringen av programmet.

Parser hanterar syntaxanalys här listan itereras igenom och baserat på en uppsättning regler matchar listans tokens till specifika delar, exemplevis en klass eller funktion. När Parsern har gått igenom hela listan kommer den att returnera ett abstrakt syntax träd, vilket representerar programmet.

För att exekvera programmet skickas det abstrakta syntax trädet till interpretatorn som traverserar trädet och evaluerar varje nod. När hela trädet har travesterats returneras resultatet av exekveringen.

3.2 Lexikalisk analys

Den lexikaliska analysen genomförs av den egen utvecklade Lexern. När en instans av klassen Lexer skapas skickas strängen med text med i konstruktorn. För att generera listan med tokens anropas funktionen tokenize. Tokenize använder sig av hjälp-funktionen next_token för att hämta nästa token som matchas i texten.

Next_token fungerar genom att bryta ut delar av strängen baserat på regler. Reglerna är uppbyggda av reguljära uttryck. Next_token går igenom strängen tills det inte finns några fler tokens att matcha och meddelar då Tokenize funktionen om detta.

Lexer kan hantera några fel som om inte varje startparantes har en slutparantes eller om ett citattecken inte har ett slut-citattecken eller om ett nummer inte är skrivet på rätt sätt. När lexer stötter på ett problem kommer den att kasta ett fel och berätta för användaren vad som är fel och vart felet är.

3.2.1 Token

Klassen token representerar en token som skapas av Lexern. Denna klassen innehåller vilken typ av token som hittas, vad värdet är, samt på vilken rad i koden som den finns på.

3.3 Syntaktisk analys

Den syntaktiska analysen genomförs av den egen utvecklade Parsern. Parsern kan klassas som en recursive descent parser. Parsern arbetar från toppen av programmet och jobbar sig neråt för att bygga upp det abstrakta syntaxträdet med hjälp av nodeklasser.

Vi körning skapas en instans av Parserklassen och funktionen produce_ast anropas med en strängen som innnehåller användarens kod. Parsern använder sedan Lexers funktion tokenize för att skapa en lista av tokens. Parsern tar listan av tokens och försöker matcha dessa till olika delar av BNF:en. När en full matchning

är hittad skapar den en relevant ast-nod för den matchningen. Sedan upprepas denna processen tills listan med tokens är slut eller ett fel uppstår.

Parsern försöker hitta alla fel som inte är beroende av vad som händer under körningen och meddelar användaren att den måste rätta till något. Några exempel på fel som parsern kan upptäcka är om en till delning till en variabel datatyp inte matchar, detta funkar dock inte med funktionsanrop då de måste evalueras först, samma sak gäller för binärauttryck. Parsern ger även fel om någon oväntad token dyker upp.

3.4 Exekvering

Exekveringen utförs av den egenutvecklade Interpretatorn. Vid körning av programmet anropas funktionen evaluate med program-noden och den nuvarande miljön. Program-noden innehåller alla andra noder som bygger upp programmet. Interpretatorn går då varje nod och evaluerar den genom att anropa den relevanta hjälpfunktion som finns i modulerna ExpressionsEvaluator eller StatementsEvaluator.

3.5 Noder

Språket byggs upp av noder. Dessa noder finns i modulen Nodes och symboliserar det abstrakta syntax trädet. Full dokumentation för alla noder finns i den autogenererade-dokumentationen. En nod kan innehålla flera olika saker, det beror på vad det är för typ av nod, men alla noder innehåller tre gemensamma saker, vilken typ av nod det är exempelvis om det är en funktionsdeklaration eller nummervärde, vilken rad denna noden finns på i användarens kod samt ett värde för noden.

Beroende på vilket uttryck parsern har matchat kommer en nod för det uttrycket skapas. För att se hur noderna kan fungera se listing 35.

```
1 int a = 2 * 3
```

Listing 35: Variabel deklaration

Här ska en variabel av typen int och namnet a skapas. Variabeln ska tilldelas värdet 2 * 3 vilket kommer bli 6. Noden som skapas i detta fallet är en VarDeclaration, denna noden tar en typ vilket kommer vara int, en identifierare som är a samt ett uttryck för vad som ska tilldelas.

Uttrycket 2 * 3 kommer att matchas mot en BinaryExpr-nod, noden tar en vänster- och högersida samt ett operator. Vänstersidan kommer att bli två, högersidan kommer bli 3 och operatorn kommer bli multiplikation(*). När noden sedan evalueras kommer BinaryExpr-noden att evalueras först, och resultatet av den evalueringen kommer tilldelas till variabeln a och variabeln kommer deklareras i den nuvarande miljön.

Genom att evaluera BinaryExpr först och använda resultatet kommer syntaxträdet att utföras med rätt prioritet. För ett mer avancerat exempel på hur prioriteringen fungerar se listing 36.

```
1 int a = 2 * 3 + 3 * 2
```

Listing 36: Variabel deklaration med prioritet

När ovanstående uttryck ska parsas kommer tre stycken binaryExpr-noder att skapas. Den första kommer vara för uttrycket 2 * 3 och den andra kommer vara för uttrycket 3 * 2. Den sista noden kommer ta den första binaryExpr-noden(2 * 3) som sin vänstrasida och den högrasidan kommer bli binaryExpr-noden(3 * 2) och operatorn kommer bli addition(+). När noden sedan ska evalueras kommer vänster- och högersidan att evalueras först och resultatet av dessa kommer att adderas och resultatet kommer bli 12 som förväntat. Oavsett hur långt uttryck som läses in kommer det att brytas ner i ett binaryExpr med en vänster- och högersida som är binaryExpr och dessa uttryck kan i sin tur också ha andra binaryExpr som sin vänster- och högersida, men då uttryckens sidor evalueras först kommer korrekt prioritet att uppnås.

3.6 Environment

Klassen Environment representerar scope i språket. Varje gång en instans skapas kan den ta emot ett Environment som blir föräldrascopet. Ett Environment innehåller tre instansvariabler, identifiers, identifier_type och constants, dessa tre är listor som innehåller informationen om de olika variabler, funktioner med mera som är deklarerade i scopet.

Environment innehåller även en klassvariabel ”global_env”, denna variabel används när de inbyggda funktionerna ska deklarerats. Skulle variabeln vara satt till något betyder det att de inbyggda funktionerna redan skapats och då kommer ett fel att kastas om att funktionerna redan har deklarerats. Det är även detta Environment som alla environments kommer ha som förälder någonstans, således blir detta det globala_scopet, allt som skapas här finns tillgängligt exakt överallt.

Environment har en funktion lookup_identifier som används för att slå upp en identifierare. Denna funktionen använder hjälpfunktionen find_scope för att hitta scopet som en identifierar finns i. Funktionen fungerar genom att den kolla om identifieraren finns deklarerad i det nuvarande scopet, skulle den inte finnas kommer anropa den find_scope på föräldra-scopet. När identifieraren är hittad kommer den instansen att returneras genom anropskedjan och komma till lookup_identifier och funktionen hämtar värdet på identifieraren och returnerar det. Å andra sidan om identifieraren inte hittas returnerars nil och lookup_identifier-funktionen kommer att kasta ett fel och meddela användaren om att identifieraren inte finns deklarerad.

Environment har även funktioner för att deklarera variabler, funktioner och klasser samt att tilldela en variabel ett nytt värde. Deklarerings-funktionerna fungerar på samma sätt, det som skiljer dem åt är vad det är de kräver och vad de sparar. Det första som händer är att funktionerna använder funktionen find_scope för att hitta om identifieraren redan finns deklarerad, om så är faller kommer ett fel kastas och användaren meddelas om att identifieraren redan är deklarerad. Finns inte identifieraren kommer den att sparas i identifierar-listan och relevant information läggs till som värdet på just den identifieraren.

Ett exempel på hur Environment används är vid en if-sats eller loop. När dessa evalueras skapas en ny environment instans som får en förälder med det nuvarnade scopet. Detta resulterar i att alla variabler som skapas inuti if-satset eller loopen kommer att försvinna efter evalueringen, men genom att de har en förälder kommer de fortfarande åt de variabler som är deklarerade utanför just den if-satsen eller loopen.

3.7 Runtime

I modulen Runtime återfinns alla relevanta klasser och funktioner för att exekverar programmet. I denna modul finns Interpretatorn och dess hjälpfunktioner, alla värden som skapas under exekveringen, alla inbyggda funktioner och Environment klassen. Interpretatorn har en metod för varje nods som existerar. Dessa metoder har ansvar för att evaluera just den ast-noden och returnera resultatet. Alla dessa metoder är privata och för att evaluera ett abstrakt-syntaxträd behöver den publika-metoden evaluate anropas. Denna metoden har som ansvar att vidarebefordra en ast-nod till korrekt metod. Om en ast-nod innehåller en eller flera andra ast-noder kommer dessa att evalueras först. När alla barn-noder till en nod har evaluerats kan noden evalueras. Detta resulterar i att de noder som är längre ner i det abstrakta-syntaxträdet kommer att evalueras först.

Eftersom Cobra är ett interpreterat språk är det i detta steget som koden körs. Vid körning kommer Cobra-koden att översättas till Ruby-kod och Rubys funktionalitet kan användas för att utvärdera Cobra-koden.

3.8 Datatyper

Alla datatyper som existerar i språket finns samlade under modulen Values och representeras av en egen klass. Det är dessa värden som skapas när programmet evalueras och det är dessa värden som användaren kan interagera med. Exempelvis när en siffra evalueras kommer en instans av NumberVal att skapas, denna instans har värdet av siffran och om det är en int eller float.

Alla datatyps-klasserna ärver från basklassen RuntimeVal. RuntimeVal-klassen implementerar alla metoder som ska vara möjliga att anropa på en datatyp. Finns det ingen bra generell lösning för hur den metoden ska bearbeta alla noder kommer istället metoden att ge ett fel och ett lämpligt felmeddelande. För att en datatyp ska kunna använda de metoder som inte har en generell lösning måste dessa överlagras i varje datatyp som kräver den funktionaliteten.

3.9 Kodstandard

I projektet har *Rubys style guide*[1] använts i en större utsträckning. Vissa avsteg har gjort som exempelvis att funktionsanrop utan parametrar anges med parenteserna istället för utan vilket de ska göras enligt guiden. Explicita returns har använts istället för att de implicita som finns i Ruby och som guiden vill att man använder. Alla kommentarer är skriva efter *Yards guide*[2] för att kunna autogenerera dokumentation för koden.

3.10 Packeteringen av kod

Koden finns att hämta på Gitlab. Där finns det en README-fil som berättar hur språket laddas ner, hur språket kan köras samt hur dokumentationen för språket kan genereras.

3.11 Tester

För att säkerställa att systemet fungerar som planerat har det testats med enhetstester. Dessa tester är skrivna i Rubys inbyggda verktyg Test::Unit. Enhetstesterna är uppdelade i tester för lexern, parsern och interpretatorn. Testerna för respektive modul är sedan grupperade i olika filer där varje fil innehåller alla tester för en specifik funktionalitet. Testerna går att köra med kommandot *rake test* som kommer köra alla tester som finns.

[1] https://github.com/rubocop/ruby-style-guide
[2] https://rubydoc.info/gems/yard/file/docs/GettingStarted.md

3.12 Grammatik

Språket är uppbyggt enligt följande BNF(Backus-Naur-form):

```
<program> ::= <stmt_list>

<stmt_list> ::= <stmt><stmt_list>
              | <stmt>

<stmt> ::= <var_declaration>
         | <conditional>
         | <func_declaration>
         | <class_declaration>
         | <assign_stmt>
         | <loop>
         | <expr>
         | <return_stmt>
         | <break_stmt>
         | <continue_stmt>

<var_declaration> ::= "const" <type_specifier | identifier> <identifier> <var_declaration_tail>
                    | <type_specifier | identifier> <identifier> <var_declaration_tail>
                    | "const" <array_type_specifier> <identifier> <array_declaration_tail>
                    | <array_type_specifier> <identifier> <array_declaration_tail>
                    | "const" <hash_type_specifier> <identifier> <hash_declaration_tail>
                    | <hash_type_specifier> <identifier> <hash_declaration_tail>

<var_declaration_tail> ::= "=" <expr>
                         | empty

<array_declaration_tail> ::= "=" <array_literal>
                           | empty

<array_literal> ::= <type_specifier>"[" <expr> <array_literal_tail> "]"
                  | <type_specifier>"[" "]"

<array_literal_tail> ::= "," <expr> <array_literal_tail>
                       | empty

<hash_declaration_tail> ::= "=" <hash_literal>
                          | empty

<hash_literal> ::= <hash_type_specifier>"{" <identifier> "=" <expr> <hash_literal_tail> "}"
                 | <hash_type_specifier>"{" "}"

<hash_literal_tail> ::= "," <identifier> "=" <expr>  <hash_literal_tail>
                       | empty

<class_declaration> ::= "Class" <identifier> "{" <class_members> "}"
                      | "Class" <identifier> "->" <identifier> "{" <class_members> "}"
```

```
<class_members> ::= <var_declaration> <class_members>
                  | <func_declaration> <class_members>
                  | empty

<func_declaration> ::= "func" <func_specifier> <identifier> "(" <func_params> ")" "{" <stmt_list> "}"

<func_params> ::= <type_specifier> <identifier> <opt_func_params>
                | empty

<opt_func_params> ::= "," <type_specifier> <identifier> <opt_func_params>
                    | empty

<func_specifier> ::= "void" | <type_specifier>

<type_specifier> ::= "int" | "float" | "bool" | "string" | <array_type_specifier>

<array_type_specifier> ::= <type_specifier> "[]"
                         | <identifier> "[]"

<hash_type_specifier> ::= "Hash<" <type_specifier> "," <type_specifier | identifier> ">"

<conditional> ::= "if" <expr> "{" <stmt_list> "}" <elsif_block> <else_block>

<elsif_block> ::= "elsif" <expr> "{" <stmt_list> "}" <elsif_block>
                | empty

<else_block> ::= "else" "{" <stmt_list> "}"
               | empty

<loop> ::= "while" <expr> "{" <stmt_list> "}"
         | "for" <var_declaration> "," <logical_expr> "," <primary_expr> "{" <stmt_list> "}"
         | "for" <identifier> "in" <identifier> "{" <stmt_list> "}"

<assign_stmt> ::= <identifier> <assign_operator> <expr>
                | <container_access> <assign_operator> <expr>

<assign_operator> ::= "+="
                    | "-="
                    | "*="
                    | "/="
                    | "="

<func_call> ::= <identifier> "(" <func_call_params> ")"

<func_call_params> ::= <expr> <opt_func_call_params>
                     | empty

<opt_func_call_params> ::= "," <expr> <opt_func_call_params>
                         | empty

<return_stmt> ::= "return" <expr>
```

```
<break_stmt> ::= "break"

<continue_stmt> ::= "continue"

<expr> ::= <logical_expr>
         | <func_call>
         | <method_call>
         | <property_call>
         | <primary_expr>

<method_call> ::= <expr> "." <func_call>

<property_call> ::= <expr> "." <expr>

<logical_expr> ::= <logical_and_expr> | <logical_or_expr>

<logical_and_expr> ::= <comparison_expr> { "&&" <comparison_expr> }

<logical_or_expr> ::= <logical_and_expr> { "||" <logical_and_expr> }

<comparison_expr> ::= <additive_expr> { <logical_comparator> <additive_expr> }

<additive_expr> ::= <multiplication_expr> { ("+" | "-") <multiplication_expr> }

<multiplication_expr> ::= <unary_expr> { ("*" | "/" | "%") <unary_expr> }

<unary_expr> ::= ("+" | "-" | "!") <primary_expr>

<primary_expr> ::= <identifier>
                 | <container_access>
                 | <numeric_literal>
                 | <boolean_literal>
                 | <string_literal>
                 | <hash_literal>
                 | <array_literal>
                 | "(" <expr> ")"
                 | <class_instance>

<class_instance> ::= "new" <identifier> "(" ")"
                   | "new" <identifier> "(" <func_call_params> ")"

<container_access> ::= <identifier> "[" <expr> "]"

<identifier> ::= /[a-zA-Z_][a-zA-Z0-9_]*/

<numeric_literal> ::= /[0-9]+ ( "." [0-9]+ )?/

<boolean_literal> ::= "true" | "false"

<string_literal> ::= '"' /.*/ '"'
```

```
                    | "'" /.*/ "'"

<logical_comparator> ::= "<" | ">" | "<=" | ">=" | "==" | "!="
```

4 Reflektion

Kollar man tillbaka på språkspecifikationen från början av projektet är det mesta implementerat, det var bara några få delar som ändrades eller fick tas bort. Från början var det tänkt att en instans av en klass skulle skapas med Class.new() men det slutade med att det istället blev new Class(). Sedan räckte inte tiden till för att lägga till publika, skyddade och privata funktioner och variabler i klasser, samt att arv för klasser inte implementerades.

Då vi båda föredrar ett lite mer strikt språk som C++ gjorde det ganska enkelt att ta fram en idé om hur språket skulle vara byggt. Vi hade även liknande tankar i hur syntaxen skulle vara således gick det snabbt att ta fram en BNF och lite kodexempel.

4.1 Albins erfarenheter

Detta projektet var riktigt roligt, är klart höjdpunkten under första året på IP. Man fick ta allt det man lärt sig och använda det för att bygga en bra produkt. Även om jag programmerat mycket tidigare så känns det som att jag har lärt mig väldigt mycket.

Det jag tar med mig från detta projektet att vi skulle ha börjat jobba tillsammans med en gång så att det inte blev att jag gjorde det mesta av parsern då jag visste hur den fungerade från början. Tanken var från början att jag skulle prova att göra en grundläggande parser innan vi skulle börja utvecklingen och se om det var något vi skulle kunna tänka oss att göra. Med tanke på att vi var trögstartade med att börja bygga språket och hamnade vi lite efter och då blev det att vi använde den parser jag byggt tidigare som grund. Då jag tyckte att det var intressant och bygga en egen så hade jag redan utökat den med flera lite mer avancerade funktioner och det började redan då bli ett ganska stort projekt. Detta ledde till att Filip fick det svårt att sätta sig in i parsern och kunna utveckla i samma takt som jag.

Från start så kändes det som att det skulle vara ganska klurigt och jobbigt att skapa mer avancerade funktioner i spårket, t.ex Klasser och funktioner men ju mer man byggde upp av de mindre delarna desto fler märkte man att man kunde återanvända senare. Klasser som jag trodde skulle ta lång tid att utveckla gick att implementera på en dag ungefär och sedan polera så att allt fungerade korrekt.

En av de jobbigaste delarna men ett av de smartaste valen vi gjorde var att skriva mycket enhetstester med en gång. Som sagt var det inte lika roligt att behöva testa samma sak om och om igen men genom att testerna fanns och att jag hade lagt till att de skulle köras vid en push till Gitlab så gjorde detta att man märkte med en gång om något man gjort tidigare gick sönder när ny funktionalitet lades till. Detta är helt klart något jag kommer forsätta att göra när jag utvecklar för det är verkligen guld värt.

4.2 Filips erfarenheter

Projektet har varit svårt för mig. Den största anledningen till det var att jag underskattade hur svårt det skulle vara att förstå en parser man själv inte varit med och byggt upp. Det tog lång tid för mig att dels förstå men också senare att använda under projektet.

Ifall jag skulle göra om projektet skulle jag varit med och gjort parsern från början så jag fått förståelse för hur den är uppbyggt och fungerar. Alternativt börjat implementeringsfasen med mycket parprogrammering så jag kunde få en bättre förståelse över parsern.