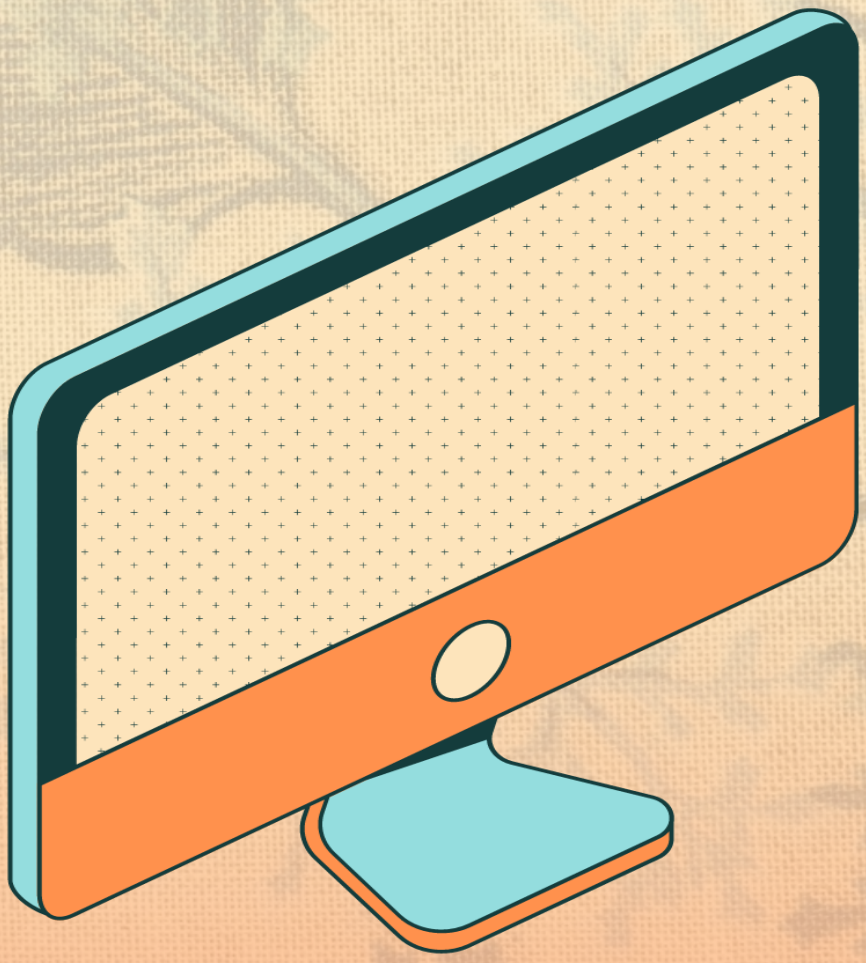
TDP019 - Projekt: Datorspråk

nial language



Alexander Engström
aleen361@student.liu.se

Institutionen för datavetenskap
Linköpings Universitet
Vårterminen 2023

LINKÖPINGS UNIVERSITET

Sammanfattning

Programspråket *Nial* har utvecklats som en del av kursen TDP019 Projekt: Datorspråk. Kursen är en del av programmet Innovativ programmering vid Linköpings universitet. Målet med projektet har varit att skapa ett nytt programspråk med hjälp av det befintliga programspråket *Ruby*.

För att utveckla *Nial* krävdes först att språkets grammatik och design definierades. Därefter implementerades kod för lexikografisk analys, semantisk analys och utvärdering av ett abstrakt syntax-träd.

Resultatet av projektet är programspråket *Nial*, ett objektorienterat interpreterat språk med en lättläst grammatik.

Innehåll

1 Revisionshistorik

Version	Beskrivning	Datum
1.0	Första versionen av dokumentet	2023-05-07

2 Inledning

Detta dokument beskriver programspråket *Nial* som utvecklades som en del av kursen TDP019 Projekt: Datorspråk under den andra terminen av programmet Innovativ programmering vid Linköpings universitet. *Nial* är ett objektorienterat interpreterat språk med en lättläst grammatik.

Dokumentet kommer att börja med en användarhandledning, där språkets datatyper, konstruktioner och användningsområden kommer att beskrivas. Därefter beskrivs språkets design samt dess grammatik med BNF-notation. Dokumentet innehåller också en systemdokumentation som beskriver systemet från ett tekniskt perspektiv. Till sist innehåller dokumentet reflektioner och diskussioner kring arbetsprocessen.

2.1 Syfte

Syftet med projektet har varit att utforska hur programspråk fungerar och hur de är uppbyggda genom att utveckla ett eget programspråk.

2.2 Målgrupp

Målgruppen för språket *Nial* är nybörjare. Språkets grammatik är lättläst och använder få förkortningar vilket gör mindre komplexa program lätta att förstå. Mer avancerade program kan ge motsatt effekt då raderna riskerar att bli långa och röriga.

Eftersom *Nial* är ett interpreterat språk är språket ineffektivt jämfört med många andra språk, vilket också är en anledning till att *Nial* inte bör användas till avancerade program.

3 Användarhandledning

I detta avsnitt beskrivs språkets funktionalitet och hur språket kan användas för att skapa program. I avsnittet kommer en del termer att användas som behöver definieras från början för att underlätta förståelsen av användarhandledningen. "Tal" refererar till en instans av typen *Number*, vilket beskrivs i avsnitt 3.3.1. "Textsekvens" refererar till en instans av typen *Text*, vilket beskrivs i avsnitt 3.3.2. "Sanningsvärde" refererar till en instans av typen *Boolean*, vilket beskrivs i avsnitt 3.3.3. "Lista" refererar till en instans av typen *List*, vilket beskrivs i avsnitt 3.4.1. "Uppslagsverk" refererar till en instans av typen *Dictionary*, vilket beskrivs i avsnitt 3.4.2. "Verktyg" refererar till datatypen *Utility*, vilket beskrivs i avsnitt 3.14.

3.1 Installation

För att komma igång med användningen av språket behöver kodbiblioteket [1] laddas ned.

I kodbiblioteket finns installationsfilen *nial/SETUP.sh* som kan köras för att genomföra installationen automatiskt. Detta kan göras via terminalen med följande kommande:

```
bash SETUP.sh
```

Installationsfilen kommer att utföra fyra operationer:

- Installerar Ruby om Ruby inte redan är installerat på det aktuella systemet.
- Skapar ett Bash-alias som möjliggör körning av Nial-kod på ett enklare sätt.
- Installerar nial-mode för Emacs.
- Genererar aktuell dokumentation av kodbasen.

När installationen är färdig kan *Nial*-kod köras genom följande kommando i terminalen oavsett katalog:

```
nial example.nial
```

Det är också möjligt att köra *Nial*-kod i interpreterat läge utan extern fil. För att starta *Nial*-interpretatorn räcker det med följande kommando i terminalen:

```
nial
```

Om installationsfilen inte används kan *Nial*-kod fortfarande köras på följande sätt:

```
ruby source/nial.rb example.nial
```

Det är viktigt att notera att sökvägen *source/nial.rb* enbart fungerar om programmet körs från projektets huvudkatalog. Om ett program skall köras från en annan katalog måste den relativa sökvägen till filen *nial.rb* anges istället. Detta gäller enbart om installationsfilen ej har körts.

[1] `https://github.com/alexandengstrom/nial`

3.2 Hello world

Ett vanligt exempel inom programspråk är att skapa ett program som skriver ut texten *"Hello world"* till terminalen. I detta avsnitt beskrivs därför hur detta går till i språket *Nial*.

Det första steget är att skapa en ny fil, i detta exempel används filnamnet *helloworld.nial*. Detta kan göras via terminalen med kommandot:

```
touch helloworld.nial
```

Öppna sedan denna fil med en textredigerare. *Emacs* rekommenderas som textredigerare då stöd för syntax-färgläggning finns om installation har genomförts enligt instruktionerna i avsnitt 3.1.

Fyll sedan filen med följande kod:

```
display ``hello world''
```

När filen är sparad kan koden köras genom att skriva följande kommando i terminalen:

```
nial helloworld.nial
```

Programmet bör nu skriva ut *"Hello world"* till terminalen.

3.3 Primitiva datatyper

I detta avsnitt beskrivs de primitiva datatyper som finns i språket. Dessa är *Number, Text, Boolean, Null*. Primitiva datatyper initialiseras inte som övriga typer som beskrivs i 3.13.2. Hur de primitiva datatyperna initialiseras beskrivs i detta avsnitt.

En primitiv datatyp är den minsta beståndsdelen av data som kan existera i programspråket. Alla andra datatyper kommer på ett eller annat sätt vara en sammansättning av primitiva datatyper.

Alla datatyper har ett antal metoder. Vad en metod är och hur dessa funkar finns beskrivet i sektion 3.13.4. I detta avsnitt kommer metoderna beskrivas övergripande.

De metoder som finns tillgänliga för samtliga primitiva datatyper är:

- **copy()**: Returnerar en kopia på objektet.
- **type()**: Returnerar en textsekvens som beskriver objektets typ.

3.3.1 Number

Number är en datatyp som representerar ett tal. Detta kan vara både ett heltal eller ett decimaltal. Ett tal skapas genom att skriva en tal.

Exempel:

```
x = 1
```

I detta exempel tilldelas variabeln x en instans av typen *Number* med värdet 1. Mer information om hur variabeltilldelning fungerar kommer i avsnitt 3.6.

Om ett decimaltal ska definieras används en punkt före decimalerna på följande vis:

```
x = 3.14
```

Det är även möjligt att definiera negativa tal genom att använda den unära operatorn - framför talet. Exempel:

```
x = - 3
```

Metoder

Ingen av de metoder som finns tillgänliga för datatypen *Number* modifierar objektet. Dessa metoder kommer istället att returnera nya objekt och lämnar det aktuella talet oförändrat. Nedan listas de metoder som finns tillgängliga för instanser av typen *Number*:

- **convert_to_text()**: Konverterar talet till en textsekvens.
- **convert_to_boolean()**: Konverterar talet till ett sanningsvärde. Alla tal förutom 0 konverteras till *True*.
- **square_root()**: Beräknar kvadratroten ur talet.
- **power(num)**: Returnerar ett nytt tal som lagrar resultatet av talet upphöjt till *num*.
- **factorial()**: Returnerar ett nytt tal som lagrar fakulteten av talet.
- **round(num)**: Avrundar talet till *num* decimaler.
- **absolute_value()**: Returnerar ett nytt talsom lagrar absolutbeloppet av talet.
- **cosine()**: Returnerar ett nytt tal som lagrar cosinus av talet.
- **sine()**: Returnerar ett nytt tal som lagrar sinus av talet.

Operatorer

I tabellerna 3.1, 3.2, 3.3 och 3.4 beskrivs vilka operatorer som kan användas tillsammans med datatypen *Number*.

Tabell 3.1: Aritmetiska operatorer som kan användas med datatypen *Number*

Operator	Argument	Returnerar	Exempel	Beskrivning
+	Number	Number	2 + 3 = 8	Adderar två tal
-	Number	Number	5 - 3 = 2	Subtraherar två tal
*	Number	Number	2 * 3 = 6	Multiplicerar två tal
*	Text	Text	2 * “x” = “xx”	Upprepar texten *n* gånger
/	Number	Number	6 / 2 = 3	Dividerar två tal
%	Number	Number	10 % 3 = 1	Returnerar resten efter heltalsdivision

Tabell 3.2: Jämförelseperatorer som kan användas med datatypen *Number*

Operator	Argument	Returnerar	Exempel	Beskrivning
==	Number	Boolean	2 == 3 = False	Jämför om två tal är samma tal. Om det andra argumentet inte är ett tal kommer sanningsvärdet *False* alltid att returneras.
!=	Number	Boolean	2 != 3 = True	Jämför om två tal inte är samma tal. Om det andra argumentet inte är ett tal kommer sanningsvärdet *True* alltid att returneras.
>	Number	Boolean	2 > 1 = True	Kontrollerar om det första talet är större än det andra
>	Text	Boolean	2 > “text” = False	Kontrollerar om det första talet är större än längden på texten
<	Number	Boolean	2 < 1 = False	Kontrollerar om det första talet är mindre än det andra
<	Text	Boolean	2 < “text” = True	Kontrollerar om det första talet är mindre än längden på texten
>=	Number	Boolean	2 >= 2 = True	Kontrollerar om det första talet är större eller lika med än det andra
>=	Text	Boolean	2 >= “text” = False	Kontrollerar om det första talet är större än eller lika med längden på texten
<=	Number	Boolean	2 <= 2 = True	Kontrollerar om det första talet är mindre eller lika med än det andra
<=	Text	Boolean	2 <= “text” = True	Kontrollerar om det första talet är mindre än eller lika med längden på texten

Tabell 3.3: Logiska operatorer som kan användas med datatypen *Number*

Operator	Argument	Returnerar	Exempel	Beskrivning
and	Alla	Boolean	0 and “text” = False	Konverterar båda datatyperna till sanningsvärden och jämför om båda är sanna
or	Alla	Boolean	0 or “text” = True	Konvererar båda datatyperna till sanningsvärden och kontrollerar om ett av dessa är sanna
not		Boolean	not 3 == False	Konverterar talet till ett sanningsvärde och inverterar det

Tabell 3.4: Tilldelningsoperatorer som kan användas med datatypen *Number*

Operator	Argument	Returnerar	Exempel	Beskrivning
+=	Number	Number	variabel += 2	Adderar 2 till variabeln
-=	Number	Number	variabel -= 2	Subtraherar 2 från variabeln
/=	Number	Number	variabel /= 2	Tilldelar variabeln resultatet av division mellan det första talet och det andra
*=	Number	Number	variabel *= 2	Tilldelar variabeln produkten av multiplikation mellan det första och det andra talet

3.3.2 Text

Text är en datatyp som representerar text. Detta kan vara ett helt dokument av text, en bokstav eller ingen text alls. Ett instans av typen *Text* skapas genom att omge en serie tecken med dubbla citattecken (“). I detta exempel tilldelas variabeln *x* värdet “hello world”:

```
x = ``hello world''
```

En textsekvens kan också innehålla specialtecken som nyradstecken (`\n`). Exempel:

```
x = ``hello\nworld''
x => hello
world
```

Eftersom dubbla citattecken används för att definiera typen kan dubbla citattecken ej användas inuti sekvensen. För att kunna inkludera dubbla citattecken inuti en sekvens måste escape-tecken användas. Escape-tecken är ett sätt att ändra betydelsen av tecknet genom att placera ett *backslash* framför tecknet. Exempel:

```
x = ``this is an \"escaped\" sequence''
x => this is an ``escaped'' sequence
```

För att hämta en specifik bokstav eller ett tecken ur en textsekvens kan indexoperatorn [] användas. Indexoperatorn kräver ett tal som argument inom hakparenteserna för att identifiera positionen av det önskade tecknet. Detta tal kallas index, och det första tecknet i sekvensen har index 1. Genom att använda indexoperatorn med rätt index returneras ett tecken. Exempel:

```
x = ``hello world''
x[2] => e
```

På samma sätt är det möjligt att omdefiniera ett tecken på ett speciellt index genom att använda hakparenteser. Exempel:

```
x = ``hello world''
x[2] = ``a''
x => hallo world
```

Om index-operatorn används med ett tal mindre än 1 eller med ett tal större än längden på textsekvensen kommer felet *ListIndexOutOfRange* returneras. Om en annan datatyp än *Number* används som argument kommer datatypen att konverteras till ett tal. Om konverteringen misslyckas kommer felet *ConversionError* att returneras.

Metoder

- **convert_to_boolean()**: Konverterar textsekvensen till ett sanningsvärde. En tom textsekvens är *False*, alla andra sekvenser är *True*.
- **convert_to_number()**: Försöker konvertera textsekvensen till ett tal. Om detta misslyckas returneras felet *ConversionError*.
- **convert_to_list()**: Konverterar textsekvensen till en lista där varje tecken i textsekvensen blir ett element i listan.
- **length()**: Returnerar ett tal som lagrar antalet tecken i textsekvensen.
- **upper()**: Returnerar en kopia av textsekvensen där alla tecken är i versaler.
- **lower()**: Returnerar en kopia av textsekvensen där alla tecken är i gemener.
- **strip()**: Returnerar en kopia av textsekvensen utan eventuella blanktecken i början och slutet.
- **left_justify(num)**: Returnerar en kopia av textsekvensen. Om sekvensen är kortare än *num* tecken adderas blanktecken till slutet av sekvensen så att längden på sekvensen blir *num*.
- **right_justify(num)**: Returnerar en kopia av textsekvensen. Om sekvensen är kortare än *num* tecken adderas blanktecken till början av sekvensen så att längden på sekvensen blir *num*.
- **sub(pattern, value)**: Returnerar en kopia av textsekvensen där alla delsekvenser som matchar *pattern* har bytts ut mot *value*. Parametern *pattern* skall vara ett reguljärt uttryck.
- **split_lines()**: Delar upp textsekvensen efter nyradstecken och returnerar en instans av typen *List* där varje rad i textsekvensen blir ett element i listan.
- **split_words()**: Delar upp textsekvensen efter samtliga blanktecken och returnerar en instans av typen *List* där varje ord eller tecken i textsekvensen blir ett element i listan.
- **split(pattern)**: Delar upp textsekvensen efter alla delsekvenser som matchar *pattern*. Returnerar en instans av typen *List* där alla delsekvenser mellan de delsekvenser som matchade *pattern* blir ett element i listan. Parametern *pattern* ska vara ett reguljärt uttryck.

Operatorer

I tabellerna 3.5, 3.6, 3.7 och 3.8 beskrivs vilka operatorer som kan användas tillsammans med datatypen *Text*.

Tabell 3.5: Aritmetiska operatorer som kan användas med datatypen *Text*

Operator	Argument	Returnerar	Exempel	Beskrivning
+	Text	Text	“hello “ + “world” = “hello world”	Konkatenerar två textsekvenser
*	Number	Text	“x” * 2 = “xx”	Upprepar textsekvensen *n* gånger

Tabell 3.6: Jämförelseoperatorer som kan användas med datatypen *Text*

Operator	Argument	Returnerar	Exempel	Beskrivning
==	Text	Boolean	“hello” == “world” = False	Jämför om två textsekvenser är identifiska
==	Number	Boolean	“hello” == 5 = True	Jämför om längden på textsekvensen är lika med *n*
==	Boolean	Boolean	“hello” == False = False	Konverterar textsekvensen till ett sanningsvärde och jämför sedan om sanningsvärdena är samma
!=	Text	Boolean	“hello” != “world” = True	Jämför om två textsekvenser inte är identifiska
!=	Number	Boolean	“hello” != 5 = False	Jämför om längden på textsekvensen inte är lika med *n*
!=	Boolean	Boolean	“hello” != False = True	Konverterar textsekvensen till ett sanningsvärde och jämför sedan om sanningsvärdena inte är samma
>	Text	Boolean	“hello” > “world” = False	Jämför om längden på den första textsekvensen är längre än den andra
>	Number	Boolean	“hello” > 2 = True	Jämför om längden på den första textsekvensen är större än *n*
<	Text	Boolean	“hello” < “world” = False	Jämför om längden på den första textsekvensen är kortare än den andra
<	Number	Boolean	“hello” < 2 = False	Jämför om längden på den första textsekvensen är mindre än *n*
>=	Text	Boolean	“hello” >= “world” = True	Jämför om längden på den första textsekvensen är längre än eller lika med den andra
>=	Number	Boolean	“hello” >= 2 = True	Jämför om längden på den första textsekvensen är större än eller lika med *n*
<=	Text	Boolean	“hello” <= “world” = True	Jämför om längden på den första textsekvensen är mindre än eller lika med den andra
<=	Number	Boolean	“hello” <= 2 = False	Jämför om längden på den första textsekvensen är mindre än eller lika med *n*

Tabell 3.7: Logiska operatorer som kan användas med datatypen *Text*

Operator	Argument	Returnerar	Exempel	Beskrivning
and	Alla	Boolean	“text” and True = True	Konverterar båda datatyperna till sanningsvärden och jämför om båda är sanna
or	Alla	Boolean	“” or 0 = False	Konvererar båda datatyperna till sanningsvärden och kontrollerar om ett av dessa är sanna
not		Boolean	not 3 == False	Konverterar textsekvensen till ett sanningsvärde och inverterar det

Tabell 3.8: Tilldelningsoperatorer som kan användas med datatypen *Number*

Operator	Argument	Returnerar	Exempel	Beskrivning
+=	Text	Text	variabel += “text”	Konkatenerar *variabel* med “text”
*=	Number	Text	variabel *= 2	Upprepar textsekvensen i *variabel* *n* gånger

3.3.3 Boolean

Boolean är en datatyp som representerar ett sanningsvärde. En instans av typen *Boolean* kan enbart anta ett av två möjliga värden, *True* eller *False*.

Ett sanningsvärde kan definieras genom att skriva *True* eller *False*. Det är viktigt att att den första bokstaven är en versal. Exempel:

```
x = True
```

Ett sanningsvärde kan också definieras genom en operation som returnerar ett sanningsvärde. Exempel:

```
x = 1 > 2
x => False
```

Sanningsvärden används i villkorssatser och villkorsbaserad upprepning för att avgöra vad programmet skall göra. Villkorssatser beskrivs i avsnitt 3.12 och villkorsbaserad upprepning beskrivs i avsnitt 3.11.3. Sanningsvärden kan också vara en lämplig datatyp för en variabel som enbart kan anta ett av två möjliga värden.

Metoder

- **convert_to_number()**: Konverterar sanningsvärdet till ett tal med värdet 0 eller 1.

Operatorer

I tabellerna 3.9 och 3.10 beskrivs vilka operatorer som kan användas tillsammans med datatypen *Boolean*.

Tabell 3.9: Jämförelseoperatorer som kan användas med datatypen *Boolean*

Operator	Argument	Returnerar	Exempel	Beskrivning
==	Alla	Boolean	True == “text” = True	Konverterar argumentet till ett sanningsvärde och jämför om de har samma värde
!=	Alla	Boolean	True != “text” = False	Konvererar argumentet till ett sanningsvärde och jämför om de inte har samma värde

Tabell 3.10: Logiska operatorer som kan användas med datatypen *Boolean*

Operator	Argument	Returnerar	Exempel	Beskrivning
and	Alla	Boolean	True and “text” = True	Konverterar argumentet till ett sanningsvärde och kontrollerar om båda är sanna
or	Alla	Boolean	True or “text” = True	Konvererar argumentet till ett sanningsvärde och kontrollerar om ett av dessa är sanna
not		Boolean	not True	Inverterar sanningsvärdet

3.3.4 Null

Null är en datatyp i språket som används för att representera avsaknaden av ett värde. Datatypen *Null* returneras automatiskt från alla funktioner och metoder som inte har något returvärde.

Metoder

- **convert_to_boolean()**: Returnerar alltid en instans av typen *Boolean* med värdet False.

Operatorer

Objekt av datatypen *Null* kan enbart använda logiska operatorer. I tabell 3.11 förklaras dessa.

Tabell 3.11: Logiska operatorer som kan användas med datatypen *Null*

Operator	Argument	Returnerar	Exempel	Beskrivning
and	Alla	Boolean	Null and True = False	Konvererar båda objekten till sanningsvärden och kontrollar om båda är sanna
or	Alla	Boolean	Null or True = True	Konvererar båda objekten till sanningsvärden och kontrollerar om ett av dessa är sant
not		Boolean	not Null	Returnerar alltid sanningsvärdet True

3.4 Sammansatta datatyper

Sammansatta datatyper är en metod för att lagra en samling av data som är relaterade till varandra. I programspråket *Nial* finns det två sammansatta datatyper: *List* och *Dictionary*. Dessa beskrivs i följande avsnitt. Även användardefinierade typer kan betraktas som sammansatta datatyper men dessa kommer att beskrivas i ett separat i avsnitt 3.13 på grund av deras omfattande funktionalitet.

3.4.1 List

En lista utgör en sammansättning av element, varvid varje element kan tillhöra vilken datatyp som helst. Ett element kan också vara en ny lista som i sin tur innehåller fler element. En lista är ett bra sätt att lagra en samling av element som har en relation till varandra. En lista definieras genom att ange ett godtyckligt antal element innanför hakparenteser. Exempel:

```
list = [1, 2, 3, 4, 5]
```

Det är också möjligt att skapa en tom lista genom att enbart ange hakparenteser. Exempel:

```
list = []
```

Detta kan vara användbart om listan kommer fyllas med element senare. Till exempel i en upprepningstruktur.

Det är möjligt att hämta ut ett element ur listan genom att använda index-operatorn. Exempel:

```
list = [1, 2, 3, 4, 5]
list[1] => 1
```

Metoder

- **convert_to_boolean()**: Returnerar sanningsvärdet *False* om listan är tom, annars returneras sanningsvärdet *True*.

- **length()**: Returnerar antalet element i listan som ett tal.
- **append(value)**: Lägger till elementet *value* i listan. Elementet kommer placeras längst bak i listan.
- **pop()**: Tar bort det sista elementet ur listan samt returnerar det värdet.
- **has_value(value)**: Returnerar sanningsvärdet *True* om listan innehåller elementet *value*. Annars returneras sanningsvärdet *False*.
- **first()**: Returnerar det första elementet i listan.
- **last()**: Returnerar det sista elementet i listan.
- **join(separator)**: Konverterar listan till en textsekvens med parametern *separator* mitt emellan alla element i textsekvensen.
- **flip()**: Returnerar en ny lista som är reverserad.
- **empty()**: Returnerar sanningsvärdet *True* om listan är tom, annars returneras sanningsvärdet *True*.
- **clear()**: Tömmer listan på element.
- **insert(index, value)**: Sätter in objektet *value* på plats *index*.
- **delete_index(index)**: Tar bort elementet på plats *index* ur listan.
- **slice(start, stop)**: Returnerar en ny lista som innehåller elementen som låg på index mellan index *start* och index *stop* i listan.

Operatorer

I tabell 3.12 beskrivs vilka operatorer som kan användas på object av typen *List*.

Tabell 3.12: Aritmetiska operatorer som kan användas med datatypen *List*

Operator	Argument	Returnerar	Exempel	Beskrivning
+	Alla	List	[1, 2, 3] + 4 = [1, 2, 3, 4]	Returnerar en ny lista som innehåller elementet i den ursprungliga listan plus argumentet.

3.4.2 Dictionary

Dictionary är en datatyp som lagrar nyckel-värde-par. Nyckeln måste vara av typen *Text* eller *Number*, medan värdet kan vara av vilken datatyp som helst. Värdet kan också vara en annan instans av typen *Dictionary*.

Ett uppslagsverk skapas genom att använda måsvingar. Innanför måsvingarna kan ett godtyckligt antal nyckel-värde-par anges. Kolon används för att separare nyckel och värde. Komma används för att separera nyckel-värde-par. Exempel:

```
dictionary = {``one'': 1, ``two'': 2}
```

När ett uppslagsverket är skapat kan indexoperatorn användas för att hämta ett värde ur uppslagsverket.

Exempel:

```
dictionary = {``one'': 1, ``two'': 2}
dictionary[``one''] => 1
```

Det går också att lägga till ett nytt värde till uppslagsverket genom att använda indexoperatorn. Exempel:

```
dictionary = {``one'': 1, ``two'': 2}
dictionary[``three''] = 3
dictionary => {``one'': 1, ``two'': 2, ``three'': 3}
```

Om nyckeln redan existerar när tilldelningsoperatorn används kommer istället värdet för denna nyckel att omdefinieras.

Metoder

- **convert_to_boolean()**: Returnerar sanningsvärdet *False* om uppslagsverket inte innehåller några nyckel-värde-par, annars returneras sanningsvärdet *True*.
- **convert_to_list()**: Returnerar en lista där varje element i listan är en lista som innehåller två element. Nyckeln och värdet.
- **get_keys()**: Returnerar en lista innehållandes alla nycklar i uppslagsverket.
- **get_values()**: Returnerar en lista innehållandes alla värden i uppslagsverket.
- **get_pairs()**: Returnerar en lista där varje element i listan är en lista som innehåller två element. Nyckeln och värdet.
- **has_key(key)**: Returnerar sanningsvärdet *True* om uppslagsverket innehåller nyckeln *key*. Annars returneras sanningsvärdet *False*.
- **has_value(value)**: Returnerar sanningsvärdet *True* om uppslagsverket innehåller värdet *value*. Annars returneras sanningsvärdet *False*.
- **length()**: Returnerar antalet nyckel-värde-par i uppslagsverket.
- **clear()**: Tar bort samtliga nyckel-värde-par från uppslagsverket.

3.5 Operatorer

Samtliga operatorer samt deras prioritet och associativitet hos finns beskrivet i tabell 3.13. Alla operatorer går inte att använda med alla datatyper. Vilka operatorer som kan användas med vilka datatyper beskrivs i avsnitten 3.3 och 3.4.

Tabell 3.13: Samtliga operatorer i programspråket *Nial*

Prioritet	Operator	Beskrivning	Associativitet
1	a[b]	Indexoperator	Vänster till höger
1	not a	Inverterar ett sanningsvärde	Höger till vänster
1	- a	Unärt minus	Höger till vänster
1	+ a	Unärt plus	Höger till vänster
2	a@b	Hämtar attribut b ur typen a	Vänster till höger
3	a^b	a upphöjt till b	Höger till vänster
4	a * b	Multiplication	Vänster till höger
4	a / b	Division	Vänster till höger
4	a % b	Modulus	Vänster till höger
5	a + b	Addition	Vänster till höger
5	a - b	Subtraktion	Vänster till höger
6	a == b	Jämför likhet mellan värden	Vänster till höger
6	a != b	Jämför olikhet mellan värden	Vänster till höger
6	a > b	Jämför om ett värde är större än ett annat	Vänster till höger
6	a < b	Jämför om ett värde är mindre än ett annat	Vänster till höger
6	a >= b	Jämför om ett värde är större än eller lika med ett annat	Vänster till höger
6	a <= b	Jämför om ett värde är mindre än ett annat	Vänster till höger
7	a and b	Kräver att båda uttrycket är sanna	Vänster till höger
7	a or b	Kräver att ett av uttrycken är sant	Vänster till höger
8	a += b	Additionstilldelning	Höger till vänster
8	a -= b	Subtraktionstilldelning	Höger till vänster
8	a *= b	Multiplikationstilldelning	Höger till vänster
8	a /= b	Divisionstilldelning	Höger till vänster
9	a = b	Variabeltilldelning	Höger till vänster

3.6 Variabler

En variabel är en identifierare som lagrar ett värde. Som namnet variabel antyder kan värdet som identifieraren lagrar ändras under programmets gång. I vissa programspråk behöver variabler deklareras innan de kan användas. Det behövs inte i *Nial*. Det räcker med att definiera variabeln. Vilken datatyp variabeln ska lagra behöver inte heller definieras då vilken datatyp en variabel lagrar också kan ändras under programmets gång.

En identifierare måste börja med gemen och kan därefter innehålla versaler, gemener, siffror och understreck.

En variabel definieras genom en identifierare följt av tilldelningsoperatorn och därefter det värde variabeln ska tilldelas. Exempel:

```
x = 2
```

När en variabel definieras sparas en referens till objektet som variabeln ska lagra. Det innebär att flera variabeler kan lagra samma objekt. Exempel:

```
x = 1
y = x
y += 1
```

I detta exempel tilldelas först talet 1 till variabel x. Sedan tilldelas variabel y värdet av variabel x, vilket är talet 1. Till sist adderas talet 1 till det tal som variabel y lagrar, vilket är samma tal som variabel x lagrar. Detta innebär att både x och y lagrar talet 2 efter dessa rader kod.

Om detta inte är det beteende som önskas kan metoden *copy* användas som samtliga primitiva och sammansatta datatyper har. Exempel:

```
x = 1
y = call x.copy()
y += 1
```

Eftersom värdet som variabel x lagrar blir kopierat lagrar inte variabel x och variabel y samma objekt i detta fall. Det innebär att det enbart är värdet som y lagrar som blir adderat med 1.

3.6.1 Konstanter

En konstant fungerar på samma sätt som en variabel med den viktiga skillnaden att värdet på en konstant aldrig kan ändras. En konstant definieras genom att använda en identifierare som inleds med två versaler och därefter innehåller endast versaler, siffror och understreck. En konstant bör användas istället för en variabel om värdet som ska lagras inte ska ändras under programmets gång. Detta gör det tydligare att förstå att detta värde inte ska ändras men det säkerställer också att värdet inte ändras av misstag.

Om ett värde definieras som en konstant kommer alla metoder och operationer som förändrar objektet att förbjudas. I exemplet nedan skulle felet *ConstantAlreadyDefined* returneras:

```
CONSTANT = 2
CONSTANT += 2
```

Det är inte heller tillåtet att tilldela ett nytt värde till identifieraren. Det innebär att även detta exempel skulle returnera felet *ConstantAlreadyDefined*:

```
CONSTANT = 2
CONSTANT = 3
```

Det är däremot tillåtet att använda alla operatorer och metoder som enbart returnerar nya objekt. Exempel:

```
LIST = [1, 2, 3, 4, 5]
new_list = LIST + 6
```

I detta exempel kommer variabeln *new_list* lagra en lista som innehåller alla element som konstanten *LIST* lagrade samt talet 6. Konstanten *LIST* lagrar fortfarande listan *[1, 2, 3, 4, 5]* och därför är dessa operationer tillåtna.

Om en användardefinierad typ initialiseras som konstant kommer samtliga attribut i typen att bli konstanta för den instansen av typen. Det är tillåtet att tilldela värden till variabler via konstruktorn men därefter kan inga attribut förändras. Om ett attribut i typen definieras som konstant är det inte heller tillåtet att tilldela ett värde via konstruktorn. Notera att en typ som inte definieras som konstant kan ha konstanta attribut.

3.7 Funktioner

En funktion är en datatyp i språket som lagrar ett block av kod. En funktion måste definieras en gång och kan sedan anropas godtyckligt många gånger. Att använda funktioner är ett bra sätt att undvika kodrepetition.

3.7.1 Funktionsdefinition

En funktion definieras genom att använda nyckelordet *define function* följt av namnet på funktionen och sedan vilka parametrar funktionen ska ta inom parenteser. Därefter följer funktionskroppen som sedan avslutas med nyckelordet *stop*.

I detta exempel skapas en funktion som tar två argument och returnerar summan av dessa:

```
define function add(x, y)
    return x + y
stop
```

3.7.2 Parameteröverföring

Alla argument som skickas med till en funktion skickas per automatik som referens. Det innebär att det är möjligt att modifiera objektet inuti funktionen och detta kommer påverka variabler som håller detta objekt utanför funktionen. Om detta inte är det beteende som önskas så kan nyckelordet *copy* användas vid funktionsdefinitionen. Om nyckelordet *copy* används framför parameternamnet kommer denna parameter att kopieras varje gång funktionen anropas.

I detta exempel kommer parametern x att kopieras men inte y:

```
define function func(copy x, y)
```

När nyckelordet *copy* används anropas metoden *copy* som finns definierad i alla inbyggda datatyper. Det kommer inte att fungera för användardefinierade typer om inte metoden *copy* överlagras i typ-definitionen. Hur detta går till beskrivs i avsnitt 3.13.7.

3.7.3 Returvärde

I en funktion kan nyckelordet *return* användas för att antingen returnera ett värde eller avbryta funktionsanropet. Om nyckelordet *return* används utan något värde efteråt eller om funktionen saknas nyckelordet *return* kommer funktionen att returnera en instans av typen *Null*.

När nyckelordet *return* påträffas inuti en funktion kommer funktionsanropet att avbrytas och ingen mer kod inuti funktionen kommer evalueras.

3.7.4 Funktionsanrop

För att kalla på en funktion används nyckelordet *call* följt av funktionsnamnet. Till sist anges de argument som ska skickas med till funktionen inom parenteser. Parenteserna måste anges även om funktionen inte tar några argument.

Exempel:

```
call func(x, y)
```

3.7.5 Anonyma funktioner

En anonym funktion fungerar som en vanlig funktion men har några begränsningar. En anonym funktion kan enbart ha ett uttryck i sin funktionskropp och det uttrycket returneras automatiskt utan att nyckelordet *return* behöver användas.

En anonym funktion definieras på följande vis:

```
| x | x % 2 == 0 |
```

Denna funktion tar en siffra som argument och returnerar *True* om talet är jämnt och *False* om talet är udda.

Det går även att direkt kalla på en anonym funktion genom att använda nyckelordet *call* samt skicka med argument inom parenteser:

```
call | x | x % 2 == 0 |(2)
=> True
```

Även om dessa funktioner kallas för anonyma funktioner är det möjligt att namnge dessa genom att tilldela en variabel en anonym funktion:

```
func = | x | x % 2 == 0 |
```

Det är nu möjligt att kalla på funktionen *func* på samma sätt som med vanliga funktioner.

Det rekommenderade användningsområdet för anonyma funktioner är att använda dessa i högre gradens funktioner. Exempelvis går det att använda anonyma funktioner för att filtrera en lista. I detta exempel filtreras listan så att den nya listan enbart innehåller jämna tal med hjälp av en anonym funktion.

```
define function filter(func, list)
    new_list = []
    for every item in list
        if call func(item) then
            call new_list.append(item)
        stop
    stop
    return new_list
stop

list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

call filter(| x | x % 2 == 0 |, list)
=> [2, 4, 6, 8, 10]
```

3.8 In- och utmatning

För att interagera med användaren genom terminalen kan det inbyggda verktyget *IOManager* användas. Eftersom att skriva till terminalen och att hämta data från terminalen är vanliga operationer finns det genvägar för att hantera detta. Dessa beskrivs i detta avsnitt. Fullständig information av verktyget *IOManager* finns i avsnitt 3.15.7.

Anledningen till att dessa kortkommandon finns är att göra det enkelt för någon som inte har programmerat mycket förut att förstå vad som sker.

3.8.1 Inmatning från terminalen

En vanlig operation är att be användaren om inmatning från terminalen och tilldela resultatet av detta till en variabel. Detta kan göras genom verktyget *IOManager* på detta vis:

```
x = call IOManager.input()
```

Exakt samma operation kan också utföras på detta vis:

```
let user assign x
```

3.8.2 Utskrift till terminalen

Att skriva ut ett värde till terminalen är också en vanlig operation, därför finns en genväg även för detta. Att skriva till terminalen via *IOManager* kan göras på följande vis:

```
call IOManager.output("hello world")
```

Exakt samma operation kan också utföras på detta vis:

```
display "hello world"
```

3.9 Filhantering

Att skriva till filer och läsa från filer går att göra genom verktyget *FileManager* som är en del av standardbiblioteket. Detta verktyg beskrivs i avsnitt 3.15.6.

Att läsa från en fil är en vanlig operation och därför finns det en genväg för detta med en mer lättläst grammatik. Exempel:

```
load ``file.txt'' into data
```

I detta exempel hämtas innehållet i filen *file.txt* till en textsekvens som sparas i variabeln *data*. Denna kod kommer anropa metoden *read* i verktyget *FileManager*.

Exempel på samma operation genom verktyget *FileManager*:

```
data = call FileManager.read(``file.txt'')
```

3.10 Importera kod

Vid skapande av stora program kan det vara en fördel att dela upp koden i olika filer för att skapa en bättre struktur i projektet. Det finns därför stöd i språket för att göra detta. För att inkludera en annan fil i den nuvarande filen används nyckelordet *use* följt av en textsekvens som innehåller namnet på den fil som ska inkluderas. All kod som finns i den andra filen kommer köras i det nuvarande scopet.

En fil kan inte inkluderas två gånger. Om filen som ska inkluderas redan har blivit inkluderad kommer kodraden att ignoreras. Detta gäller dock enbart om filen har blivit inkluderad i ett scope som fortfarande finns kvar. Om en fil exempelvis skulle inkluderas inuti en funktion, så kommer filen kunna inkluderas igen utanför funktionen.

Sökvägen till filen är alltid relativ till den filen som använder nyckelordet *use*. Det innebär att om båda filerna finns i samma katalog så räcker det med att ange filnamnet, oavsett från vilken katalog programmet körs.

Exempel:

```
use "testfil.nial"
```

Om filen som ska inkluderas inte finns i samma katalog anges sökvägen relativt. I detta exempel ligger filen “testfil.nial” i katalogen “test” som existerar ett steg bakåt i katalogstrukturen:

```
use "../test/testfil.nial"
```

3.11 Upprepningsstrukturer

Språket innehåller tre olika sorters upprepningsstrukturer. Dessa är intervallbaserad upprepning, listbaserad upprepning och villkorsbaserad upprepning.

I samtliga upprepningsstrukturer kan nyckelordet *break* användas för att avbryta upprepningen och nyckelordet *next* kan användas för att hoppa till nästa iteration.

3.11.1 Intervallbaserad upprepning

Intervallbaserad upprepning används för att iterera genom ett intervall av tal.

En intervallbaserad upprepningsstruktur definieras på vis:

```
count x from 1 to 10
    display x
stop
```

I detta exempel är x styrvariabeln som kan användas inuti kodblocket. I den första iterationen kommer x hålla värdet 1. I den andra iterationen kommer x hålla värdet 2. Detta fortsätter till och med den sista iterationen då x kommer hålla värdet 10. Namnet på styrvariabeln behöver inte vara x, detta är enbart ett exempel. Samma regler gäller för detta variabelnamn som för vanliga variabelnamn.

Det är också möjligt att räkna baklänges om *from* > *to*.

3.11.2 Listbaserad upprepning

Listbaserad upprepning kan användas för att iterera igenom samtliga element i en lista. Detta fungerar på liknande sätt som i intervallbaserad upprepning men styrvariabeln kommer hålla ett element ur listan i taget. I en listbaserad upprepningsstruktur kommer antalet iterationer vara samma som antalet element i listan.

Exempel:

```
for every item in list
    display item
stop
```

Listbaserad upprepning fungerar inte enbart för object av typen *List*. Det fungerar även för alla typer som har metoder *convert_to_list* definierad. Exempelvis kommer ett objekt av typen *Text* att konverteras till en lista där varje bokstav i textsekvensen blir ett element i listan. Det är också möjligt att i användardefinierade typer definiera metoden *convert_to_list*. Vid iterering över ett objekt som inte kan konverteras till en lista kommer felet *ConversionError* att returneras.

Vid listbaserad upprepning kommer styrvariabeln per automatik att vara en referens till elementet i listan. Det är möjligt att istället begära en kopia av elementet genom att använda nyckelordet *copy* framför variabelnamnet.

Exempel:

```
for every copy item in list
    display item
stop
```

3.11.3 Villkorsbaserad upprepning

Vid villkorsbaserad upprepning evalueras ett block kod så länge ett villkor är sant. Denna typ av upprepning är att föredra om det inte är känt på förhand hur många iterationer som kommer behövas.

En villkorsbaserad upprepning definieras på följande vis:

```
x = 1
while x < 10
    display x
    x += 1
stop
```

I detta exempel kommer kodblocket upprepas nio gånger.

Resultatet av villkoret kommer automatiskt att konvereras till datatypen *Boolean*. Om metoden *convert_to_boolean* inte finns definierad i den aktuella datatypen kommer felet *ConversionError* att returneras.

3.12 Villkorssatser

En villkorssats kan användas för att låta programmet köra olika kodblock beroende på ett sanningsvärde. En villkorssats måste alltid ha minst ett villkor och ett tillhörande kodblock. En villkorssats kan därefter ha godtyckligt många fler villkor med tillhörande block. En villkorssats kan också ha ett eller inget block som kommer köras om inget av villkoren var sanna. En villkorssats måste alltid avslutas med nyckelordet *stop*.

Exempel:

```
if x < 0 then
  display ``It is negative''
else if x > 0 then
  display ``It is positive''
else
  display ``It is zero''
stop
```

I denna villkorssats kommer en av tre olika textsekvenser att skrivas ut till terminalen beroende på värdet på variabeln x.

Om villkorssatsen enbart innehåller ett villkor kan den skrivas på en rad. Exempel:

```
if x == 1 then flag = False stop
```

3.13 Typer

Det kommer inte alltid att räcka med de primitiva och sammansatta datatyperna för att representera ett visst objekt. Det går därför att skapa egna datatyper. Dessa kallas för typer och beskrivs i detta avsnitt.

3.13.1 Typdefinition

En typdefinition innehåller ett namn, en eventuell annan typ som denna typ ska ärva ifrån, ett godtyckligt antal attribut och en eller flera metoder. Anledningen till att det måste finnas minst en metod är för att en typ alltid måste ha en konstruktor. Konstruktorn är den metod som kommer köras när en instans av typen skapas. Konstruktorn skapas på samma sätt som övriga metoder som beskrivs i avsnitt 3.13.4, men måste ha namnet *constructor*.

I detta exempel skapas en typ som ska representera ett spelkort:

```
define type PlayingCard
  suit = Null
  value = Null

  define method constructor(s, v)
    suit = s
    value = v
  stop
stop
```

Denna typ heter nu *PlayingCard* och har två attribut, *suit* och *value*. Dessa attribut kommer att tilldelas värden genom konstruktorn.

3.13.2 Initialisering

För att skapa en instans av en typ används nyckelordet *create* följt av namnet på typen. Efter typnamnet ska parenteser anges. Inom parenteserna anges eventuella argument till konstruktorn. I detta exempel skapas en instans av typen PlayingCard som definierades i avsnitt 3.13.1:

```
card = create PlayingCard(``spades'', 7)
```

Det går att skapa godtyckligt många instanser av en typ och dessa kommer vara oberoende av varandra.

3.13.3 Attribut

Ett attribut är en variabel som tillhör typen. Alla attribut måste definieras innan konstruktorn i typdefinitionen. Alla variabler som enbart definieras inuti konstruktorn eller i någon annan metod kommer enbart att existera inuti den metoden.

Ett attribut kan precis som en vanlig variabel även vara en konstant. Om typen har ett konstant attribut kan detta ej tilldelas ett värde genom konstruktorn.

Det är möjligt att hämta attribut från en typ utanför typen men det är inte möjligt att tilldela ett attribut ett nytt värde utanför typen. Detta måste göras via en "setter"-metod.

För att hämta ett attribut utanför typen används attribut-operatorn. Exempel:

```
card@value
```

3.13.4 Metoder

En metod är en funktion som tillhör en datatyp. För att kunna anropa en metod krävs en instans av datatypen. Undantaget är metoder definierade i verktyg som kan anropas utan en instans. Verktyg beskrivs i avsnitt 3.14. På samma sätt som en funktion kan en metod returnera värden och om inget värde returneras kommer datatypen *Null* att returneras.

För att anropa en metod skrivs metodnamnet bakom objektet separerat med en punkt. Exempel:

```
call object.method(parameter)
```

Objektet kan vara antingen en en variabel eller ett värde. I exemplet ovan skulle *object* kunna vara en variabel som lagrar en datatyp där metoden *method* är definierad.

Om en metod anropas direkt på ett värde kan det se ut på följande vis:

```
call ``hello world''.length() => 11
```

I detta exempel skapas textsekvensen *"hello world"* och sedan anropas direkt metoden *length* som returnerar längden på textsekvensen.

En metod har tillgång till alla attribut som är definierade i typen och kan modifiera attributen om de inte är konstanter.

3.13.5 Arv

I detta exempel definieras klassen *PlayingCard* med exakt samma funktionalitet som den hade i exemplet i avsnitt 3.13.1 men genom att använda arv istället:

```
define type BaseCard
  value = Null

  define method constructor(v)
    value = v
  stop
stop

define type PlayingCard
  extends BaseCard
  suit = Null

  define method constructor(v, s)
    call parent(v)
    suit = s
  stop
stop
```

Om en metod med samma namn finns definierad i både *BaseCard* och *PlayingCard* kommer metoden som finns definierad i *PlayingCard* att anropas. Det är dock möjligt att i anropa metoden i basklassen genom att anropa funktionen *parent* som konstruktorn i *PlayingCard* gör i exemplet ovan.

3.13.6 Operatoröverlagring

Om operatorn addition skulle användas på objektet som skapades i avsnitt 3.13.2 skulle felet *InvalidOperator* returneras. Det beror på att *Nial* inte vet hur två spelkort ska adderas med varandra. Detta gäller även alla andra operatorer som existerar i språket. För att kunna använda operatorer på användardefinierade typer måste operatorerna överlagras. Detta går att göra genom att definiera metoder med förutbestämda namn. Samtliga metoder behöver ta ett argument vilket är den andra operanden.

Om två spelkort ska kunna adderas genom att addera deras värde men behålla färgen från den första operanden kan följande metod definieras:

```
define method addition(other)
  new_value = value + other@value
  return create Card(suit, new_value)
stop
```

Här är det viktigt att tänka på att det är inte säkert att *other* kommer vara av typen *Card*. Skulle den andra operanden vara av typen *Number* skulle felet *VariableNotDefined* returneras då datatypen *Number* inte har något attribut som heter *value*. Det går att hantera detta genom att använda metoden *type*. Exempel:

```
define method addition(other)
  if call other.type() != ``Card'' then
    display ``Invalid operator''
    return
  stop

  new_value = value + other@value
  return create Card(suit, new_value)
stop
```

Det är viktigt att notera att den typen som överlagrar en operator som måste vara en första operanden i operationen för att det skall fungera. Om *mytype* är en instans av en typ där additionsoperatorn är överlagrad skulle följande exempel vara tillåtet:

```
mytype + 1
```

Det skulle dock inte gå att utföra denna operation:

```
1 + mytype
```

Detta beror på att det är *addition*-metoden i datatypen *Number* som anropas i detta fall och den kan inte hantera addition med en användardefinierad typ.

Nedan följer namnen på de metoder som kan användas för att överlagra en specifik operator.

- **addition(other)**: Överlagrar additions-operatorn.
- **subtraction(other)**: Överlagrar subtraktionsoperatorn.
- **multiplication(other)**: Överlagrar multiplikations-operatorn.
- **division(other)**: Överlagrar divisions-operatorn.
- **modulos(other)**: Överlagrar modulos-operatorn.
- **power(other)**: Överlagrar exponent-operatorn.
- **equals(other):** Överlagrar likhets-operatorn.
- **not_equals(other)**: Överlagrar olikhets-operatorn.
- **greater_than(other)**: Överlagrar större-än-operatorn.
- **less_than(other)**: Överlagrar mindre-än-operatorn.
- **equals_or_greater_than(other)**: Överlagrar större-än-eller-lika-med-operatorn.
- **equals_or_less_than(other)**: Överlagrar mindre-än-eller-lika-med-operatorn.
- **addition_assignment(other)**: Överlagrar lägg-till-och-tilldela-operatorn.
- **subtraction_assignment(other)**: Överlagrar subtrahera-och-tilldela-operatorn.
- **multiplication_assignment(other)**: Överlagrar multiplicera-och-tilldela-operatorn.
- **division_assignment(other)**: Överlagrar dividera-och-tilldela-operatorn.

3.13.7 Standardmetoder

Det finns också ett antal metoder som har en speciell betydelse i språket.

- **convert_to_list()**: Denna metod anropas automatiskt när ett objekt försöker användas som bas för en listbaserad upprepning. Det innebär att det är möjligt att definiera denna metod och på så vis göra det möjligt att iterera genom objektet. För att detta ska fungera måste metoden returnera en lista.
- **convert_to_boolean()**: Denna metod anropas automatiskt när ett objekt används i en villkorssats. Detta gäller även vid villkorsbaserad upprepning. Genom att definiera metoden går det att definiera när objektet har sanningsvärdet *True* och när det har sanningsvärdet *False*.
- **display()**: Denna metod anropas automatiskt när ett objekt ska skrivas ut till terminalen. Genom att definiera denna metod går det att definiera vad som ska skrivas ut till terminalen. Metoden måste returnera en textsekvens.

3.14 Verktyg

Ett verktyg är en variation av en typ med några viktiga skillnader.

Det går inte att skapa en instans av ett verktyg. När verktyget är definierat finns det en instans av detta verktyg och alla metoder kan anropas genom denna instans. Den instansen som skapas kommer ha samma namn som namnet på verktyget. Ett verktyg kan därför inte ha någon konstruktor.

En annan viktig skillnad är att ett verktyg enbart kan ha konstanta attribut. Det innebär att ett verktyg aldrig kan ändras efter att det har definierats.

Ett verktyg kan användas för att samla funktionalitet som relaterar till varandra på ett ställe för att skapa bättre struktur i programmet. Exempelvis går det att skapa ett verktyg med namnet *Math* där alla metoder som utför matematiska beräkningar kan samlas.

Ett annat användningsområde för verktyg är det som i många andra programspråk kallas *Enum-klasser*. Det innebär att det går att definiera en uppräkning av konstanta värden med ett gemensamt tema. Ett exempel skulle kunna vara om programmet behöver hålla koll på veckodagar. Ett verktyg skulle då kunna definieras på följande vis:

```
define utility Day
  MONDAY = 0
  TUESDAY = 1
  WEDNESDAY = 2
  THURSDAY = 3
  FRIDAY = 4
  SATURDAY = 5
  SUNDAY = 6
stop
```

Med detta verktyget definierat blir koden mer läsbar och enklare att förstå. Om funktionen *get_day()* returnerar aktuell dag och koden ska göra olika saker beroende på vilken dag det är kan verktyget användas på följande vis:

```
day = call get_day()
if day == Day@MONDAY then
  # do something
else if day == Day@TUESDAY then
  # do something else
stop
```

3.14.1 Verktygsdefinition

Ett verktyg defineras på samma sätt som typer definieras men nyckelordet *utility* används istället för nyckelordet *type*. I detta exempel defineras verktyget med namnet *Math*:

```
define utility Math
  PI = 3.14

  define method factorial(number)
    # factorial algorithm
  stop
stop
```

Det går nu att använda verktygets attribut och metoder. Om en area ska beräknas kan vi använda attributet *PI* på följande vis:

```
radius = 2
area = radius * radius * Math@PI
```

Det går också att anropa metoden *factorial* på följande vis:

```
result = call Math.factorial(5)
```

3.15 Standardbibliotek

Utöver den grundläggande funktionalitet som har beskrivits i avsnittet finns också ett standardbibliotek. Standardbiblioteket består av typer och verktyg som kan användas i alla projekt. Standardbiblioteket inkluderas automatiskt.

3.15.1 Canvas

Canvas är en datatyp som kan rendera grafik till terminalen. Typen *Canvas* kan exempelvis användas för att skapa enklare spel.

Färger definieras genom textsekvenser. Följande färger kan användas:
“black”, “red”, “green”, “blue”, “magenta”, “cyan”, “white”, “pink”, “orange”, “light_blue”, “coral”, “purple”, “teal”, “gold”, “lavender”, “salmon”, “sky_blue”, “rose”, “lime_green”, “dark_red”, “turquoise”, “olive”, “navy”, “mauve”, “peach”, “maroon”, “eggplant”, “mustard”.

Metoder

- **constructor()**: Skapar en ny instans av typen *Canvas*.
- **set_dimensions(width, height)**: Definierar storleken på fönstret som kommer renderas till terminalen.
- **set_pixel(x, y, color)**: Sätter pixeln med koordinaterna x = x, y = y till färgen *color*.
- **add(x, y, height, width, color)**: Skapar ett nytt *CanvasObject*, vilket beskrivs i avsnitt 3.15.2. Objektets startposition kommer vara koordinaterna x = x, y = y. Objektets höjd kommer vara *height* och objektets bredd kommer vara *width*. Hela objektet kommer också att ha färgen *color*. Metoden returnerar det *CanvasObject* objekt som skapas.

- **update()**: Skapar en buffer för nästa bild som ska renderas till terminalen.
- **fill(color)**: Fyller hela fönstret med färgen *color*.
- **render()**: Renderar den nuvarande buffern till terminalen.
- **inside(object)**: Returnerar sanningsvärdet True om objektet *object* befinner sig inuti fönstret. Annars returneras sanningsvärdet *False*.

3.15.2 CanvasObject

Ett *CanvasObject* är ett objekt som existerar i fönstret som skapas genom datatypen *Canvas*. Ett *CanvasObject* objekt kan enbart skapas genom metoden *add* som är definierad i typen *Canvas*.

Metoder

- **move(dx, dy)**: Flyttar objektet med hastigheten delta x (*dx*) och delta y *dy*.
- **get_position()**: Returnerar en lista med två element. Det första elementet är objektets x-koordinat och det andra elementet är objektets y-koordinat.
- **set_position(x, y)**: Flyttar objektet till koordinaterna *x* och *y*.
- **intersects_with(other)**: Returnerar sanningsvärdet *True* om objektet kolliderar med objektet *other*. Annars returneras sanningsvärdet *False*.

3.15.3 Database

Database är en datatyp som representerar en databas. Datatypen kan användas i program där data behöver sparas permanent. Det är enbart möjligt att spara textsekvenser och tal i en databas.

Metoder

- **constructor()**: Skapar en nytt objekt av typen *Database*.
- **connect(name)**: Ansluter till databasen med namnet *name*. Om det inte existerar en databas med det namnet kommer en sådan att skapas.
- **create_table(name, columns)**: Skapar en ny tabell i databasen med namnet *name*. Parametern *columns* skall vara en lista där varje element i listan är namnet på en kolumn i tabellen.
- **insert_into(name, values)**: Lägger till en rad i tabellen *name* med värdena *values*. Parametern *values* skall vara en lista som innehåller elementen som skall läggas till i tabellen. Antalet element måste vara lika med antalet kolumner som skapades i metoden *create_table*.
- **select(name, conditions, sort_by)**: Returnerar en lista som innehåller uppslagsverk. Parametrarna *conditions* och *sort_by* är valfria. Om dessa parameterar inte används returneras alla rader i tabellen *name*. Om parametern *condition* används skall denna parameter vara ett uppslagsverk med regler för vilka rader som skall hämtas. Till exempel skulle {“name”: “John.*”} hämta alla rader där kolumnen namn börjar på “John”. Parametern *sort_by* skall vara en textsekvens med namnet på kolumnen som resultatet skall sorteras på.
- **delete(name, condition)**: Raderar alla rader från tabellen *name* som stämmer överens med upplagsverket *condition*.
- **drop_table(name)**: Raderar tabellen med namnet *name*.

3.15.4 TimeStamp

TimeStamp är en datatyp som representerar en tidpunkt. Ett object av typen *TimeStamp* skapas genom att anropa metoden *now* i verktyget *TimeManager*. Datatypen lagrar antalet sekunder som passerat sedan den första januari 1970.

Metoder

- **convert_to_text()**: Returnerar en textsekvens som beskriver datum och tidpunkt för den aktuella tiden.
- **convert_to_number()**: Returnerar antalet sekunder som datatypen lagrar en ett tal.

3.15.5 SystemManager

SystemManager är ett inbyggt verktyg som gör det möjligt att interagera med operativsystemet. Verktyget *SystemManager* kan också användas för att hämta argument som skickats till ett program via terminalen.

Attribut:

- **ARGUMENTS**: En lista som innehåller alla kommandoradsargument.
- **PLATFORM**: En textsekvens som beskriver vilket operativsystem som används.

Metoder:

- **execute(command)**: Kör kommandot *command* i terminalen.
- **exit()**: Avslutar programmet.
- **wait(time)**: Låter programmet vänta *time* sekunder.

3.15.6 FileManager

FileManager är ett inbyggt verktyg som förenklar hantering av filer.

Metoder:

- **read(filename)**: Läser innehållet från filen *filename* och returnerar innehållet som en textsekvens. Kan också returnera felet *ConversionError* om parametern *filename* är av fel datatyp eller felet *FileError* om filen inte kan läsas.
- **write(filename, data)**: Skriver innehållet i parametern *data* till filen *filename*. Denna metod skriver över befintlig data i filen *filename*. Metoden kan också returnera felen *ConversionError* eller *FileError*.
- **append(filename, data)**: Skriver innehållet i parametern *data* till slutet av befintligt innehåll i filen *filename*. Metoden kan också returnera felen *ConversionError* eller *FileError*.

3.15.7 IOManager

IOManager är ett inbyggt verktyg som hanterar in och utmatning till programmet.

Metoder:

- **input()**: Stannar programmet och låter användaren mata in data via terminalen. Metoden returnerar det användaren har matat in som en textsekvens.
- **output(value)**: Skriver ut innehållet i parametern *value* till terminalen.
- **get_key()**: Returnerar den knapp på tangentbordet som är intryckt vid tillfället som metoden anropas. Om ingen knapp är intryckt returnerar *Null*.

3.15.8 RandOpManager

RandOpManager är ett inbyggt verktyg som hanterar slumpmässiga operationer. Detta verktyg kan användas när slump behöver användas i ett program.

Metoder:

- **get_integer(start, stop)**: Returnerar ett slumpmässigt heltal mellan *start* och *stop*.
- **get_float(start, stop)**: Returnerar ett slumpmässigt decimaltal mellan *start* och *stop*.
- **pick_item(list)**: Returnerar ett slumpmässigt element från listan *list*.

3.15.9 TimeManager

TimeManager är ett inbyggt verktyg som kan användas för generera en tidpunkt.

Metoder:

- **now()**: Returnerar ett objekt av typen *TimeStamp* med tidpunkten för när metoden anropades.

3.16 Felhantering

Om något går fel vid körning av ett program kommer en feltyp att genereras. Det finns 19 olika feltyper. Om ett fel uppstår kommer körningen av programmet att avbrytas och felet som har uppstått kommer beskrivas i terminalen. Det är möjligt att undvika detta genom att använda felhantering.

Om nyckelordet *try* används är det möjligt att fånga eventuella fel som uppstår i det kodblocket. Det går att fånga antingen ett specifikt fel eller alla eventuella fel. Alla felhanteringsblock måste avslutas med nyckelordet *stop*.

I detta exempel fångas enbart fel av typen *FileError*:

```
try
  result = call FileManager.read(``filename.txt'')
catch FileError
  display ``Failed to read file''
stop
```

I detta exempel kommer innehållet i filen *"filename.txt"* att sparas i variabeln *result* om operationen lyckas. Om operationen misslyckas med feltypen *FileError* kommer istället meddelandet *"Failed to read file"* att skrivas ut till terminalen.

Det är också möjligt att fånga alla fel genom att använda nyckelordet *else*. Exempel:

```
try
  x = 1 / 0
catch DivisionByZero
  display ``Cannot divide by zero!''
else
  display ``Something went wrong..''
stop
```

Det är enbart fel som uppstår i utvärderingssteget som kan fångas med felhantering. Fel som uppstår i den lexikografiska eller semantiska analysen kan inte fångas med felhantering.

3.16.1 Feltyper

I språket *Nial* existerar följande felkoder:

- **InvalidCharacterError**: Detta fel kan uppstå i den lexikografiska analysen av koden om det finns ett okänt tecken i koden.
- **UnexpectedTokenError**: Detta fel kan uppstå i den semantiska analysen av koden om felaktig grammatik har använts.
- **InvalidOperator**: Detta fel uppstår när en operator används för en datatyp där denna operator inte är definierad.
- **DivisionByZero**: Detta fel uppstår vid division där nämnaren är 0.
- **VariableNotDefined**: Detta fel uppstår när en variabel som inte är definierad används.
- **ConstantNotDefined**: Detta fel uppstår när en konstant som inte är definierad används.
- **ConstantAlreadyDefined**: Detta fel uppstår om en konstant modifieras.
- **WrongNumberOfArguments**: Detta fel uppstår när felaktigt antal argument skickas till en funktion eller metod.
- **ConversionError**: Detta fel uppstår när en datatyp inte kan konverteras till den datatyp som förväntades.
- **ReturnError**: Detta fel uppstår när nyckelordet *return* används utanför en funktion eller metod.
- **BreakError**: Detta fel uppstår när nyckelordet *break* används utan för en upprepningsstruktur.
- **NextError**: Detta fel uppstår när nyckelordet *next* används utanför en upprepningsstruktur.
- **InvalidDatatype**: Detta fel uppstår när en felaktig datatyp har använts.
- **TooMuchRecursionError**: Detta fel uppstår när programmet har för mycket rekursion.
- **ListIndexOutOfRange**: Detta fel uppstår när ett index som inte existerar hämtas ut en lista eller textsekvens.
- **DictionaryKeyError**: Detta fel uppstår när en nyckel som inte existerar hämtas ur ett uppslagsverk.
- **MethodMissing**: Detta fel uppstår när en metod som inte existerar anropas.
- **FileError**: Detta fel uppstår när det inte går att läsa eller skriva till en fil.
- **TypeNotDefined**: Detta fel uppstår när en typ som inte är definierad används.

3.17 Dokumentation

Det är möjligt att dokumentera kod i *Nial* på två olika sätt. Det finns vanliga kommentarer som beskrivs i avsnitt 3.17.1 och det finns dokumentationskommentarer som beskrivs i avsnitt 3.17.2.

3.17.1 Kommentarer

En kommentar börjar med specialtecknet brädgård och avslutas vid radbryt. En kommentar kommer aldrig att påverka körningen av ett program då dessa kommentarer kastas bort vid den lexikografiska analysen av koden. Det innebär att en kommentar kan placeras överallt.

Exempel:

```
# This is a comment
This is not a comment
This is not a comment # but this is a comment
```

Kommentarer bör användas för att beskriva kod som inte är självförklarande. Detta gör koden lättare att underhålla.

3.17.2 Automatisk generering av dokumentation

Det finns också möjlighet att skapa automatisk dokumentation genom att använda dokumentationskommentarer. Dessa kommentarer börjar med ett dollar-tecken, avslutas med ett dollar-tecken och kan sträcka sig över flera rader. En dokumentationskommentar kastas inte bort vid den lexikografiska analysen vilket gör att dessa kommentarer kan orsaka parsing-fel om de placeras på fel ställen. Dokumentationskommentar är enbart tillåtna inuti typer och verktyg på tre ställen:

1. En kommentar direkt under typnamnet som beskriver typen eller verktyget.
2. En kommentar under varje attribut som beskriver attributet.
3. En kommentar längst upp i varje metod för att beskriva metoden.

Ett exempel på hur detta kan se ut:

```
define type Child
  extends Base
  $
  Write a long description of the type Child

  This comment can span over multiple lines
  $

  first_attribute = Null
  $ This attribute is used for nothing $

  define method constructor()
    $ Describe the functioning of the constructor here $
    # some code
  stop
```

```
stop
```

Om denna fil heter *testfile.nial* kan dokumentation för denna fil genereras genom följande kommando i termialen:

```
nial document testfil.nial
```

Detta kommer skapa en ny katalog som heter *docs* i den katalogen som *testfil.nial* är placerad i. I katalogen *docs* kommer det skapas ytterligare tre kataloger, *xml*, *latex* och *html*.

- **XML**: Innehåller en XML-fil.
- **LaTeX**: Innehåller en tex-fil.
- **HTML**: Innehåller flertalet html-filer varav *index.html* är startpunkten för dokumentationen.

4 Språkspecifikation

I detta avsnitt beskrivs språkets design.

4.1 Grammatik

Nedan följer språkets grammatik i BNF-notation.

```
<program>            ::=   <statements>

<statements>         ::=   <statement> { <statements> } |
                           empty

<loop statements>    ::=   ( <statement> | ``break'' | ``next'' ) { <loop statements> } |
                           empty

<func statements>    ::=   ( <statement> | ``return'' [expr] ) { <func statements> } |
                           empty

<statement>          ::=   <control structure>
                           <type definition>
                           <utility definition>
                           <func definition>
                           <io>
                           <file read>
                           <try block>
                           <import>
                           <expr>
                           <comment>

<import>             ::=   ``use'' <expr>

<try block>          ::=   ``try'' <statements>
                           { ``catch'' <type identifier> <statements> }
                           [ ``else'' <statements> ]
                           ``stop''

<file read>          ::=   ``load'' <expr> ``into'' <identifier>

<io>                 ::=   <input> | <output>

<output>             ::=   ``display'' <expr>

<input>              ::=   ``let user assign'' <identifier>

<control structure>  ::=   <range loop> |
                           <list loop> |
                           <while loop> |
                           <if statement>

<range loop>         ::=   ``count'' <identifier> ``from'' <expression> ``to'' <expression>
                           <loop statements>
                           ``stop''

<list loop>          ::=   ``for every'' { ``copy'' }  <identifier> ``in'' <expr>
                           <loop statements>
                           ``stop''

<while loop>         ::=   ``while'' <logical expression> <loop statements> ``stop''

<if statement>       ::=   ``if'' <condition> ``then'' <loop statements>
                           { ``else if'' <condition> ``then'' <statements> }
                           [ ``else'' <statements> ]
                           ``stop''

<type definition>    ::=   ``define type'' <type identifier>
                           [``extend'' <type identifier>]
                           {var assign}
                           ``define method'' ``constructor'' ``(`` <params> ``)''
                           <func statements> ``stop''
                           {method definition}
                           ``stop''
```

```
<utility definition> ::=   ``define utility'' <type identifier>
                           { <const assign> } { <method definition> } ``stop''


<method definition>  ::=   ``define method'' <identifier> ``(`` <params> ``)''
                           <func statements> ``stop''


<func definition>    ::=   ``define function'' <identifier> ``(`` <params> ``)''
                           <func statements> ``stop''


<expr>               ::=   <expr> <assign op> <log expr> |
                           <var assign>
                           <log expr>

<var assign>         ::=   <identifier> ``='' <expr>

<const assign>       ::=   <constant> ``='' <expr>

<assign op>          ::=   ``+='' | ``-='' | ``*='' | ``/=''

<log expr>           ::=   <com expr> <log op> <log expr> |
                           ``not'' <log expr> |
                           <com expr>

<log op>             ::=   ``and'' | ``or''

<com expr>           ::=    <arit expr> <com op> <com expr> |
                            <arit expr>

<com op>             ::=    ``>'' | ``>='' | ``<'' | ``<='' | ``=='' | ``!=''

<arit expr>          ::=    <term> <arit op> <arit expr> |
                            <term>

<arit op>            ::=    ``+'' | ``-''

<term>               ::=    <factor> <term op> <term> |
                            <factor>

<term op>            ::=    ``*'' | ``/'' | ``%"

<factor>             ::=   <atom> | ``(`` <expr> ``)''

<atom>               ::=   <identifier> | <constant> | <number> | <text> | <bool> | <null> |
                           <list> | <dict> | <type init> | <func call> | <method call> |
                           <anon func> | <attr_access>

<type init>          ::=   ``create'' <type identifier> ``(`` <args> ``)''

<func call>          ::=   ``call'' ( <identifier> | <anon func> ) ``(`` <args> ``)''

<anon func>          ::=   ``|'' <params> ``|'' <expr> ``|''

<method call>        ::=   ``call'' ( <expr> | <type identifier> )
                           ( ``.'' <identifier> ``(`` <args> ``)'' ) +

<list>               ::=   ``[`` <args> ``]''

<params>             ::=   <identifier> { <params> } |
                           empty

<param>              ::=   ``,'' <identifier>

<args>               ::=   <expr> { <arg> } |
                           empty

<arg>                ::=   ``,'' <expr>

<dict>               ::=   ``{`` <dict entries> ``}''

<dict entries>       ::=   <expr> ``:'' <expr> { <dict entry> } |
                           empty

<dict entry>         ::=   ``,'' <expr> ``:'' <expr>
```

```
<constant>          ::=    /[A-Z]{2}[A-Z_0-9]*\b/
<type identifier>   ::=    /[A-Z]\w*/
<identifier>        ::=    /[a-z]\w*/
<number>            ::=    /[0-9]+/ | /[0-9]+\.[0-9]+/
<text>              ::=    /"(?:\\.|[^\\"])*"/
<bool>              ::=    ``True'' | ``False''
<null>              ::=    ``Null''
<comment>           ::=    ``#'' /.*/ ``\n'' | ``$'' /[.*|\n]*/ ``$''
```

4.2 Konvertering av datatyper

I alla fall där en viss datatyp förväntas kommer interpretatorn automatiskt att försöka konverera datatypen till den datatyp som förväntas. Detta sker genom att anropa någon av följande metoder:

- **convert_to_number()**
- **convert_to_text()**
- **convert_to_boolean()**
- **convert_to_function()**
- **convert_to_list()**
- **convert_to_dictionary()**

Dessa metoder finns definierade för samtliga datatyper men de kommer ibland att returnera felet *ConversionError*. Exempelvis går det inte att konvertera en funktion till en lista vilket innebär att felet *ConversionError* skulle uppstå om ett funktionsobjekt försöker användas som bas för en listbaserad upprepning.

Ett object av typen *Text* går däremot bra att konvertera till en lista vilket innebär att det går bra att använda ett Text-objekt som bas vid listbaserad upprepning.

Detta gäller i alla fall där en viss datatyp förväntas. Detta kan förenkla saker, men det kan också orsaka problem då det inte är säkert att ett fel uppstår även om användaren råkar göra ett misstag.

4.3 Scope

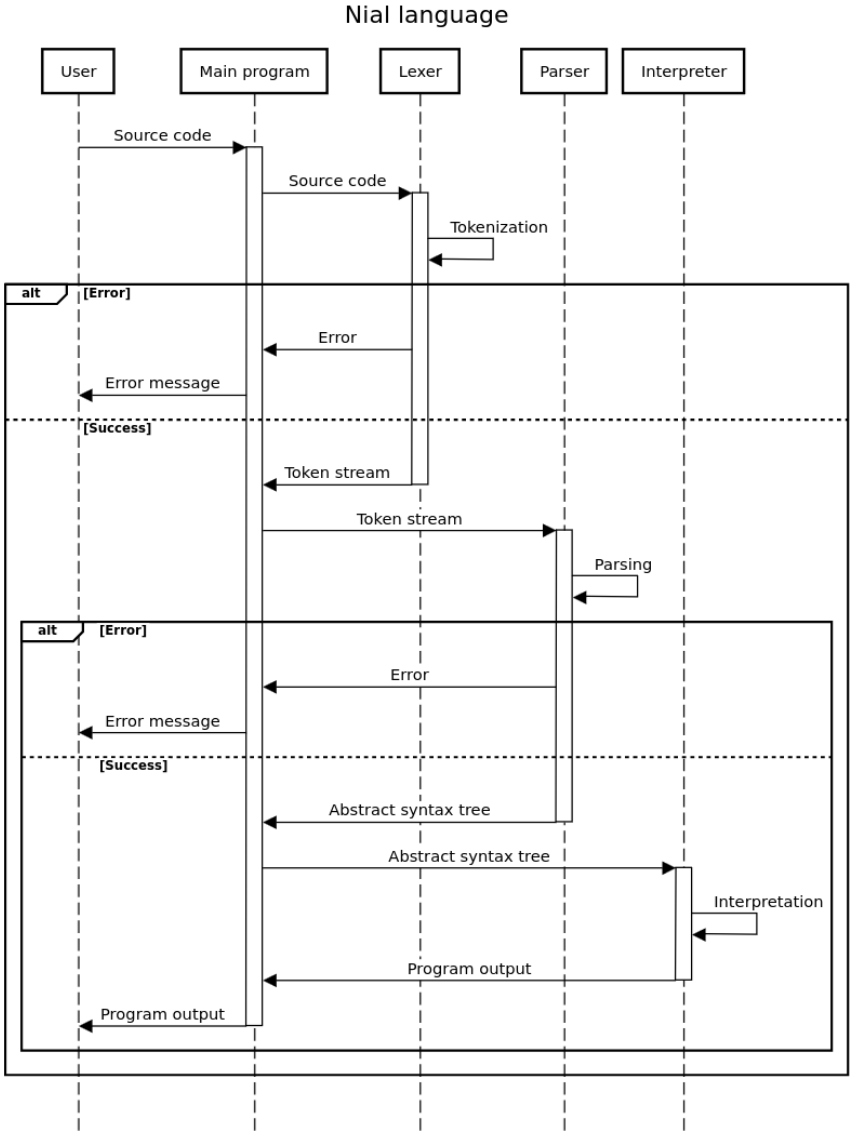
I språket finns det fyra typer av scope. Alla scope förutom det globala scopet har ett föräldra-scope. De scope som finns är *scope*, *funktions-scope*, *metod-scope* och *typ-scope*.

- **Scope**: Detta är det grundläggande scopet som gäller för det globala scopet och kontrollstrukturer. Detta scope kan alltid söka efter variabler i sitt föräldrascope.
- **Funktion-scope**: Detta scope gäller inuti funktioner. Ett funktions-scope kan enbart söka efter variabler i sitt eget scope och i det globala scopet. Det går inte att omdefiniera en variabel i det globala scopet inuti funktionen men det går att använda en variabel från det globala scopet.
- **Typ-scope**: Detta scope lagrar alla attribut och metoder hos en användardefinierad typ. Ett typ-scopes föräldrascope kommer vara det globala scopet eller det typ-scope som typen ärver av.
- **Metod-scope**: Detta scope är likt ett funktions-scope men kommer alltid ha ett typ-scope som föräldrascope. Ett metod-scope kan både använda och omdefiniera variabler i typ-scopen.

5 Systemdokumentation

I detta avsnitt beskrivs hur systemet är uppbyggt. Avsnittet kommer ge en övergripande beskrivning och inte att gå in i detalj på hur alla metoder och klasser fungerar. I kodbiblioteket [2] finns fullständig dokumentation av systemet. Denna dokumentation är i HTML-format och går att läsa genom filen *nial/docs/index.html*. Dokumentationen genereras av installationsfilen *nial/SETUP.sh*, denna fil måste därför köras innan dokumentationen kan läsas. Avsnitt 3.1 beskriver hur installationen går till.

Systemet kan delas upp i tre moduler. Lexikografisk analys, semantisk analys och evaluering. Utöver dessa finns även ett huvudprogram som knyter samman modulerna. Sekvensdiagrammet i figur 1 visar hur dessa samarbetar för att utvärdera källkod.

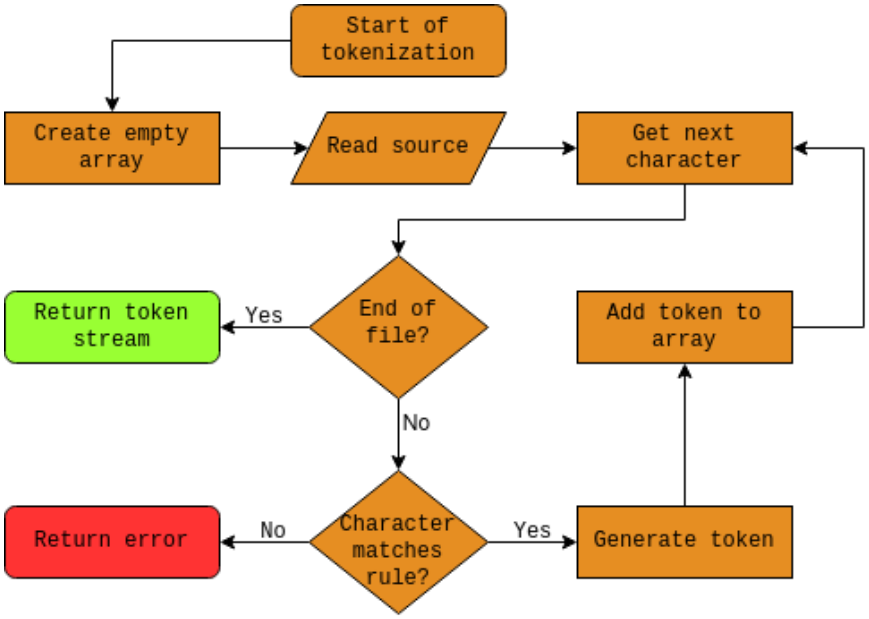


Figur 1: Visuell representation av processen för att läsa, tolka och evaluara källkod genom lexer, parser och interpreter.

[2] `https://github.com/alexandengstrom/nial`

5.1 Lexikografisk analys

Den lexikografiska analysen sker i klassen *Lexer* som tar en sträng som indata och returnerar en lista av tokens. Avsnitt 5.1.1 beskriver vad en token är. Om det finns ett tecken eller en textsekvens i indatan som inte matchar någon regel kommer felet *InvalidCharacterError* att returneras och programmet avslutas. Om all indata kan konverteras till tokens kommer en lista av dessa tokens att returneras. Figur 2 visar hur denna process går till.



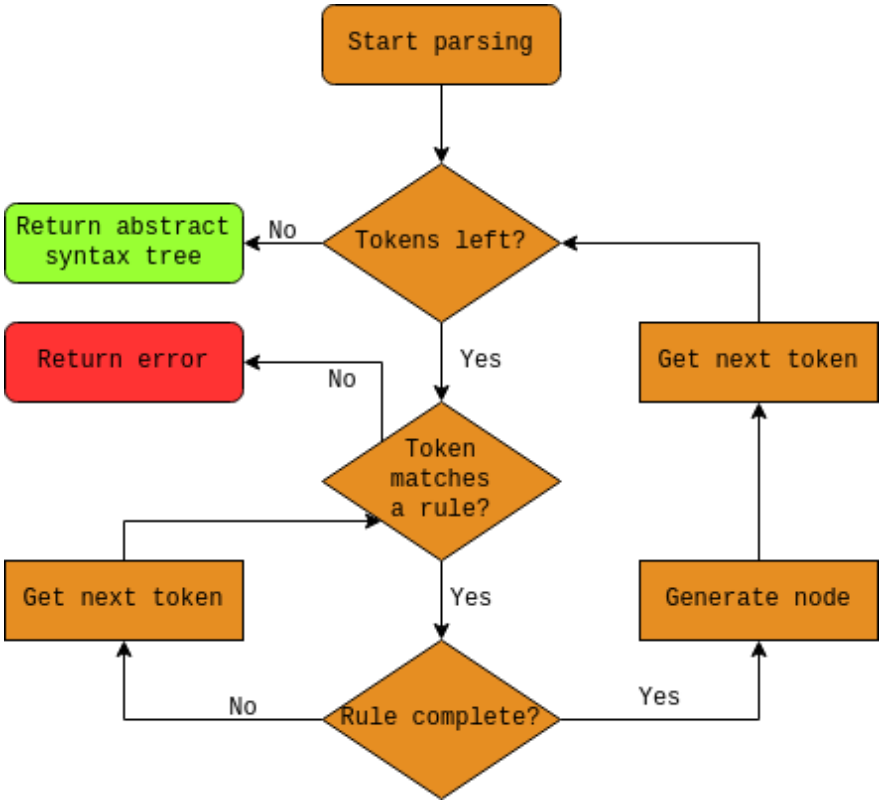
Figur 2: Flödesschema som beskriver processen att omvandla en fil med källkod till en lista av tokens.

5.1.1 Token

En token är en datastruktur som representerar en delsekvens av källkoden. I BNF-notationen i avsnitt 4.1 beskrivs vilka textsekvenser som kan bli en token. En token representeras som en instans av klassen *Token*. En token sparar information om vilken typ av token det är, det kan vara exempelvis **INT**, **ADDITION** eller **USE**. En token sparar också ett värde i de fall de är relevant. Om typen är **INT** behöver också värdet på heltalet lagras. Om typen däremot är nyckelordet **USE** behöver inget värde lagras. En token sparar också positionen där den hittades. Detta sker genom att en instans av klassen *Position* skapas som i sin tur lagrar filnamnet, raden och kolumnen. Positionen sparas för att kunna ge användaren informativa och pedagogiska felmeddelanden om varför ett fel har inträffat och vart felet har inträffat.

5.2 Semantisk analys

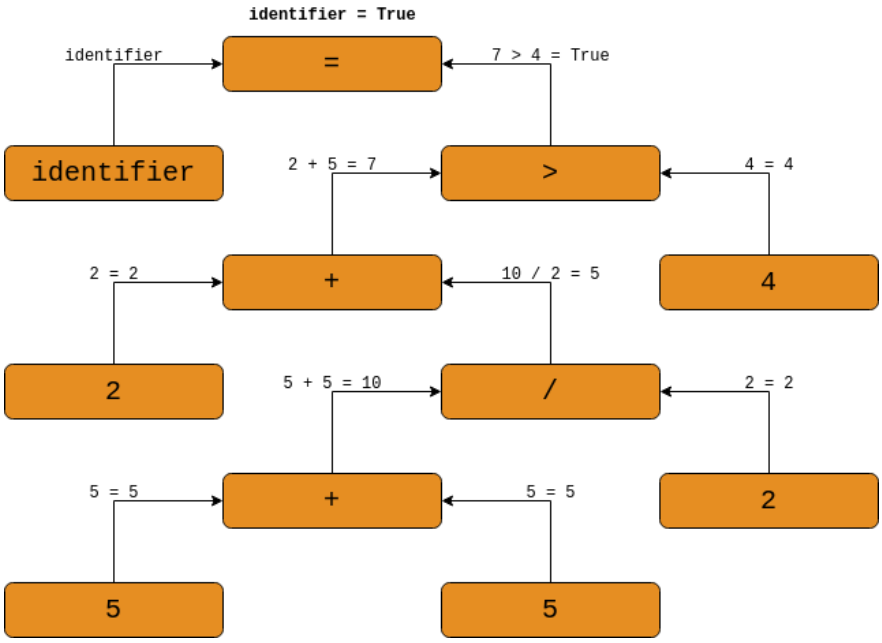
I nästa steg sker den semantiska analysen. Klassen *Parser* har ansvar för detta. Indatan i detta steg är den lista av tokens som returnerades i det förra steget. *Parser*-klassen kommer nu att försöka tolka strömmen genom att matcha en sekvens av tokens mot en regel. Även dessa regler finns definierade i avsnitt 4.1. Under tiden som listan av tokens tolkas bygger *Parser*-klassen upp en datastruktur som kallas ett abstrakt syntaxträd. Figur 3 visar hur denna process går till.



Figur 3: Flödesschema som beskriver processen att omvandla en ström av tokens till ett abstrakt syntax-träd.

Om någon del av strömmen av tokens inte matchar någon av reglerna kommer felet *UnexpextedTokenError* returneras och programmet kommer avslutas.

Det är klassen *Parser* som ansvarar för att prioritet och associativitet blir korrekt. Detta är extra viktigt i matematiska uttryck. Detta fungerar genom att exempelvis operatorer med högre prioritet hamnar längre ned i det abstrakta syntaxträdet och därför kommer evaluaras först i nästa steg. Figur 4 visar hur ett abstrakt syntaxträd kan se ut.



Figur 4: Visuell representation av syntax-träd genererat av parserutvärdering av *identifier = 2 + (5 + 5) / 2 > 4*

Om klassen *Parser* lyckas tolka hela strömmen av tokens kommer toppnoden av det abstrakta syntaxträdet att returneras.

5.2.1 Noder

Ett abstrakt syntaxträd består av ett antal noder. Noderna kan i sin tur peka mot andra noder. Exempelvis representeras alla binära operationer av klassen *BinaryOperationNode*. Denna nod har tre attribut, den vänstra noden, den högra noden och operatorn. Den vänstra och högra noden kommer vara en annan sorts nod. Det kan vara en NumericNode som enbart lagrar en siffra, men det kan också vara en annan instans av klassen *BinaryOperationNode*. Toppnoden för ett program kommer alltid vara av typen *StatementsNode*. En sådan nod lagrar en lista av alla andra noder som genererats i den semantiska analysen.

Alla noder representeras av egna klasser. Alla dessa klasser ärver från klassen *BaseNode* som lagrar en instans av klassen *Position*. Den position som sparas i en nod är den position som den första token i den aktuella regeln lagrade. Det innebär att positionen för exempelvis en *MethodCallNode* kommer vara samma position som den token med typen **CALL** hade då alla metodanrop inleds med nyckelordet *call*.

Noderna har inga metoder förutom konstruktorn. Nodernas attribut är istället publika och det är klassen *Interpreter* som ansvarar för att utvärdera noderna när den semantiska analysen är färdig.

5.3 Evaluering

Det sista steget av processen är att utvärdera det abstrakta syntaxträd som *Parser*-klassen har byggt upp. Detta sker i klassen *Interpreter*. Klassen *Interpreter* har en metod för varje sorts nod som existerar. Klassen har även en metod *evaluate* som ansvarar för att vidarebefordra en nod till en rätt metod. Om noden som skall utvärderas innehåller andra noder kommer dessa noder att utvärderas först. När alla barn-noder till en nod är utvärderade fortsätter utvärderingen av aktuella noden. Detta innebär att de noder som befinner sig längst ned i det abstrakta syntax-trädet utvärderas först, som beskrivs i figur 4.

Eftersom toppnoden av programmet alltid kommer vara en nod av typen *StatementsNode* börjar interpretatorn med att utvärdera denna nod. Beteendet för denna nod är att iterera genom samtliga barn-noder som denna nod lagrar i attributet *statements* och sedan utvärdera dessa noder. Dessa noder har också förutbestämde regler för hur de ska utvärderas vilket gör att alla noder i det abstrakta syntaxträdet utvärderas en efter en.

Eftersom *Nial* är ett interpreterat språk är det i detta steg som koden körs. Detta sker genom att översätta de konstruktioner som existerar i språket *Nial* till *Ruby*-kod. Detta gör att *Rubys* algoritmer kan användas.

5.4 Datatyper

Alla datatyper som existerar i språket representeras av en egen klass. Alla dessa klasser ärver från klassen *Null*. Anledningen till att alla klasser ärver från *Null* är för att underlätta felhantering vid användning av operatorer. Alla operatorer som kan användas för en viss datatyp måste definieras i dess klass. Detta är samma metoder som tas upp i avsnitt 3.13.6. Om en metod inte definieras för en datatyp kommer istället metoden i klassen *Null* att anropas, vilket kommer returnera felet *InvalidOperator* i de flesta fall.

Systemet är också uppbyggt på så vis att metoder som skall finnas tillgänglig för en datatyp i *Nial* kan definieras med *Ruby*-kod i datatypens klass. Detta gör det enkelt att utöka datatyperna med fler metoder. Alla metoder som skall kunna anropas från *Nial* måste alltid ta minst två arguments, scope och position. Dessa argument skickas automatiskt med från interpretatorn vid varje metodanrop, men är dolt för användaren. Dessa metodanrop felhanteras även med *Rubys* begin/rescue-block för att fånga *ArgumentError*. Detta för att undvika att systemet kraschar med *Rubys* felkoder. Istället kommer felet *InvalidNumberOfArguments* att genereras och returneras till användaren.

5.5 Felhantering

Under utvecklingen av språket har ett av målen varit att en krasch i ett program skrivet i *Nial* skall resultera i *Nials* egna felmeddelanden. För att kunna åstadkomma detta måste det säkerställas att inget av systemets beståndsdelar kraschar.

Vid alla operationer som kan orsaka fel kontrolleras detta innan operationen utförs. Om till exempel divisionsoperatorn används kommer systemet att kontrollera att nämnaren inte är noll innan operationen utförs. Om nämnaren är noll skapas och returneras istället en instans av felet *DivisionByZero*. Denna typ av kontroller genomförs på samtligen ställen där systemet skulle kunna krascha vid fel värde eller datatyp på en variabel.

Detta är ett exempel på hur datatypen valideras i metoden *round* för datatypen *Number*. I detta fall krävs ett heltal för att kunna avrunda talet korrekt. Därför valideras detta innan operationen utförs.

```
def round(number, scope, position)
  number = number.convert_to_number(scope, position)
  if number.is_a?(Error) then return number end
  return Number.new(@value.round(number.value))
end
```

Vissa typer av fel går inte att hitta genom att enbart kontrollera variabler. Vid dessa fall används istället *Rubys* begin/rescue block. Detta gäller till exempel vid läsning av fil eller om programmet har för mycket rekursion.

Om ett fel uppstår i den lexikografiska analysen kommer systemet inte gå vidare till den semantiska analysen. Huvudprogrammet kommer istället att skriva ut felet som har uppstått och sedan avsluta programmet. Detsamma gäller om ett fel uppstår i den semantiska analysen, programmet kommer då inte gå vidare till evalueringen.

5.6 Kodstandard

I projektet har kodstandarden *Ruby Style Guide*[3] använts. Vissa avsteg har gjorts från de riktlinjer som presenteras i guiden. Till exempel ska funktionsanrop vara separerade med en blankrad men i detta projekt förekommer inte blankrader mellan funktionsanrop när de är nära relaterade. Kommentarer har skapats genom standarden *RDoc*[4] för att kunna generera automatisk dokumentation.

5.7 Tester

För att säkerställa att systemet fungerar som förväntat har systemets testats på olika sätt. Dessa tester beskrivs i detta avsnitt.

5.7.1 Enhetstester

För enhetstestning har Rubys inbyggda verktyg Test::Unit använts. Enhetstesterna är uppdelade i femton olika filer. Dessa finns under *nial/tests/unittests*. För att förenkla körning av testerna finns även filen *nial/tests/run_tests.rb* som kör alla enhetstester som existerar. Filerna är uppdelade så att varje fil testar en specifik del av systemet. Till exempel så finns enhetstester för klassen *Lexer* i filen *nial/tests/unittests/test_lexer.rb* och tester för klassen *Parser* finns i filen *nial/tests/unittests/test_parser.rb*.

Totalt finns det 143 tester och 1949 påståenden som måste stämma för att systemet ska klara enhetstesterna.

5.7.2 Integrationstester

Under *nial/examples* finns tre program skrivna i *Nial* för att testa att systemet fungerar för större program. Dessa program listas nedan:

- **Snake**: Under *nial/examples/snake_game* finns ett *Snake*-spel skapat i *Nial*. Detta spel testar förutom de grundläggande strukturerna i språket även klasserna *RandOpManager*, *Canvas*, *CanvasObject*, *IOManager* och *Database* från standardbiblioteket. Spelet kan köras i terminalen genom att köra filen *nial/examples/snake_game/snake.nial*.
- **Bubblesort**: Under *nial/examples/bubblesort* finns ett exempel på bubblesort-algoritmen skriven i *Nial*.
- **Programspråk**: Under *nial/examples/language* finns ett större projekt som är ett eget programspråk i miniatyr. Detta testar framförallt att språkets typer fungerar som förväntat samt att språket kan hantera källkod från flera olika filer.

5.7.3 Effektivitetstester

Språkets effektivitet har också testats genom att köra samma kod i *Nial* och *Ruby* för att jämföra hur mycket längre tid det tar i *Nial*. Dessa tester finns under *nial/tests/speedtest*. De flesta tester visar på att *Nial* är mellan 100 och 200 gånger långsammare än *Ruby*.

[3] `https://github.com/rubocop/ruby-style-guide`
[4] `https://docs.ruby-lang.org/en/2.1.0/RDoc/Markup.html`

5.8 Brister

Det finns inga stora kända brister i språket utifrån de tester som har utförts. Det finns dock ett antal saker som skulle kunna förbättras. Dessa beskrivs i detta avsnitt.

5.8.1 Sökning i Dictionary

I de flesta programspråk är fördelen med datatypen *Hash* att det går snabbt att söka efter värden i datastrukturen. Detta gäller inte i språket *Nial*. Sökning efter nyckel-värde-par sker genom linjärsökning. Detta skulle kunna lösas genom att låta listan som lagrar nyckel-värde-paren vara sorterad och använda binärsökning istället men detta är inte implementerat.

5.8.2 Parameteröverföring

När en funktion eller metod definieras är det inte möjligt att ange ett standardvärde för en parameter som gäller om argumentet inte skickas med vid funktionsanrop. Detta är något som de flesta programspråk har stöd för men inte *Nial*.

5.8.3 Standardbiblioteket

Det saknas enhetstester för vissa klasser som är en del av standardbiblioteket. Det innebär att det mest sannolikt finns dolda problem i dessa klasser.

6 Reflektion

I detta avsnitt reflekterar jag, Alexander Engström, över projektet. Jag beskriver vad som har varit svårt, vilka lärdomar jag tar med mig och hur arbetsprocessen har sett ut. Jag reflekterar även över vad som har gått bra och vad som hade kunnat göras bättre.

6.1 Syfte och mål

Utöver kursens mål hade jag några egna mål med detta projekt. Ett av dessa var att inte använda någon annans kod. Därför valde jag att inte använda den parser vi hade tillgång till i kursen. Jag ville inte heller att språket skulle bli beroende av *Ruby*-bibliotek som inte är en del av *Rubys* standardbibliotek. Till sist ville jag att det skulle vara möjligt att skapa avancerade program i språket som till exempel enklare spel.

6.2 Utmaningar och svårigheter

Under projektets gång har jag stött på en del problem, i detta avsnitt tar jag upp de problem jag tycker var svårast att lösa.

6.2.1 Arv

Det problem jag har haft mest problem med var att kunna anropa en metod från en basklass genom att använda nyckelordet *parent*. Jag trodde att jag hade kommit på en bra lösning för detta eftersom det fungerade direkt men jag märkte snabbt att min metod inte fungerar när en klass ärver i flera steg och alla steg kallar på sin basklass.

Den slutgiltiga lösningen blev att räkna antalet metod-scope i scope-stacken med samma namn för att kunna beräkna hur många steg uppåt i klasshierakin en metod ska hämtas. Jag upplever inte lösningen som särskilt elegant men jag lyckades inte hitta något bättre sätt att lösa detta på. Jag skulle förmodligen ha behövt konstruera klasser på ett smartare sätt för att undvika detta problem.

6.2.2 Skapa ett lättläst språk

Jag trodde på förhand att det skulle vara enkelt att skapa en lättläst grammatik men jag märkte ganska snabbt att det inte var så lätt. Att inte använda förkortningar gör kanske att ett enstaka funktionsnamn blir lätt att förstå men det medför också att det snabbt blir långa rader av kod. Långa rader orsakar i sin tur att koden inte längre är så lättläst. Det är en svår avvägning som måste göras för att skapa ett lättläst språk. Jag lyckades inte riktigt med detta så bra som jag hade hoppats.

6.2.3 Felhantering

Eftersom jag ville förhindra att programmet skulle kunna krascha med *Rubys* felkoder blev det mycket arbete med att kontrollera att variabler är av rätt datatyp. Min lösning på detta blev att, om programmet förväntar sig ett tal, konverta variableln till ett tal och därefter kontrollera om operationen lyckades eller inte. Ett problem med detta är att det blir en betydande mängd metodanrop i onödan vilket förmodligen gör programmet mer långsamt. Fördelen som jag ser med denna metod är att systemet får mindre *coupling*. Det räcker att definiera exempelvis metoden *convert_to_number* ifall en datatyp skall kunna konverteras till ett tal. Annars kommer basklassen ta hand om att returnera felet *ConversionError*.

6.2.4 Relativa sökvägar

Ett problem jag inte hade tänkt på innan projektet påbörjades var hur sökvägar ska fungera. Jag insåg att jag inte vill att filer som skall inkluderas måste inkluderas relativt till den katalog från där programmet körs. Jag valde istället att alla sökvägar ska vara relativa från den fil där kodraden förekommer. Detta innebar

en del problem då interpretatorn behövde veta sökvägen till den fil koden är hämtad ifrån, vilket skedde i den lexikografiska analysen. Jag lyckades lösa detta genom att hämta filnamnet från *Position*-klassen som egentligen enbart var tänkt att användas för att skriva ut felmeddelanden. Därefter kunde jag använda *Rubys* algoritmer för att expandera sökvägen till den filen och på så vis kunna använda relativa sökvägar. Även om mina tester inte visar på några fel med denna metod så upplever jag inte att det känns helt stabilt och det kan förmodligen uppstå problem som jag inte har tänkt på.

Jag valde att använda relativa sökvägar eftersom jag tycker att programmet skall fungera likadant oavsett från vilken katalog programmet körs. De flesta programspråk har inte detta beteende som standard vilket jag tycker är lite märkligt. Det är möjligt att det finns problem med min metod som jag inte har tänkt på.

6.2.5 Grafik

Eftersom jag ville att det skulle vara möjligt att skapa enklare spel i språket behövde det också vara möjligt att rendera grafik. Eftersom jag inte ville att språket skulle vara beroende av något annat än *Rubys* standardbibliotek blev detta en utmaning. Jag undersökte vilka möjligheter som fanns inom ramen för denna kurs och jag bestämde mig för att använda terminalen. Min lösning på detta blev att generera blanktecken till terminalen samt använda escape-koder för att färglägga bakgrunden. Jag var lite skeptisk till om detta verkligen skulle fungera men det fungerade över förväntan.

6.3 Tekniska lärdomar

Jag har lärt mig mycket under projektets gång, i detta avsnitt beskriver jag de viktigaste lärdomarna.

6.3.1 Ruby

I kursen TDP007 arbetade jag med att lösa mindre problem i programspråket *Ruby* men detta blev det första stora projektet där jag har använt *Ruby* som verktyg. Från början gillade jag inte programspråket, jag upplevde att det var rörigt att arbeta med. Efter att ha arbetat med *Ruby* i några månader har jag nu en helt annan åsikt. Jag tycker att det är ett riktigt bra programspråk som är enkelt att arbeta med. Jag kommer ta med mig denna kunskap och använda *Ruby* mer i framtiden.

6.3.2 Förståelse för hur programspråk fungerar

Jag upplever att jag har fått en mycket bättre förståelse för hur ett programspråk fungerar efter att ha arbetat med detta projekt. Det är lättare att förstå varför vissa saker fungerar och vissa saker inte fungerar i andra programspråk. Det är också enklare att förstå skillnader mellan olika programspråk. Exempelvis varför vissa språk är snabbare än andra.

6.3.3 Implementera kända algoritmer

De flesta programmeringsspråk brukar skriva ut något i stil med *did you mean xxx?* om en variabel används som inte är definierad men där variabelnamnet är likt en annan variabel. Jag bestämde mig därför för att implementera detta i *Nial*. När jag läste på om detta hittade jag en algoritm som heter *Levenshtein distance* som används för att jämföra hur lika två olika strängar är. Jag hittade pseudo-kod för denna algoritm och översatte den sedan till *Ruby*-kod för att kunna använda den i mitt projekt. Det enda jag behövde ändra i algoritmen var att normalisera resultatet till ett värde mellan 0 och 1 för att kunna avgöra vilket existerande variabelnamn som är mest likt variabeln som användaren har använt.

Det jag tar med mig från detta är att det ofta finns algoritmer för generella problem och det är förmodligen smartare att använda dessa istället för att försöka tänka ut egna lösningar.

6.3.4 Enhetstester

I detta projekt var jag väldigt noga från start med att skriva mycket enhetstester för alla nya komponenter i systemet. Jag tror att detta har varit totalt avgörande för att systemet fungerar så pass bra som det gör. Jag har upptäckt extremt många märkliga fel genom mina tester som jag aldrig hade upptäckt utan enhetstesterna. Det ger också en känsla av trygghet att kunna köra samtliga tester direkt efter en modifikation av koden för att säkerställa att ändringen inte har förstört något innan nästa modifikation påbörjas.

Det som var mindre bra med enhetstesterna är strukturen på testerna. Från början upplever jag att det fanns bra struktur på testerna, men allt eftersom projektet växte blev strukturen sämre och sämre. Kommentarerna som beskriver testerna blev också sämre och sämre. Jag tar med mig att det är viktigt att ha en bra och genomtänkt struktur för enhetstesterna.

6.4 Arbetsprocessen

I tidigare projekt har jag arbetat ihop med en partner och vi har i dessa projekt nästan uteslutande par-programmerat. När detta projekt påbörjades var tanken att vi skulle använda samma upplägg. Min partner blev tyvärr tvungen att avbryta sina studiera kort efter att projektet påbörjades vilket gjorde att jag har fått arbeta med detta projekt själv. Detta blev en ny erfarenhet för mig.

Överlag har det fungerat bra att arbeta ensam. Jag har kunnat bestämma själv hur jag vill designa språket och hur jag ska lösa de problem som uppkommer. Jag har även kunnat styra vilka tider jag arbetar med projektet vilket har gjort att jag arbetat mer tidseffektivt med detta projekt jämfört med andra projekt.

Det finns också nackdelar med att arbeta själv. Många problem som jag har stött på i detta projekt hade förmodligen kunnat lösas snabbare om vi hade varit två personer som försökt lösa problemet ihop. Jag upplever också att många av de sämsta förslagen och ideérna sållas bort snabbt när två personer arbetar ihop. Det är sällan två personer tycker att en dålig idé är en bra idé. Det kan dock hända att en person tycker att dålig idé är en bra idé.

Jag tycker ändå att det har fungerat bra att arbeta med projektet och jag har fått nya erfarenheter. Jag föredrar dock fortfarande att programmera i par.

6.5 Resultatet

Överlag är jag nöjd med den slutprodukt som jag lyckades ta fram i detta projekt. Jag är framförallt nöjd med att jag lyckades implementera klasser, arv och en egen parser. Det finns mycket funktionalitet som skulle kunna läggas till i språket för att göra det mer komplett men jag måste också inse att målet med denna kurs inte är att skapa ett komplett programspråk. Jag har lärt mig mycket om hur datorspråk är uppbyggda vilket var det största målet med detta projekt.

Det jag är mest missnöjd med är att språket inte blev så lättläst som jag hade hoppats. Att balansera detta var svårare än jag hade förväntat mig.

6.6 Framtida projekt

Eftersom jag har uppskattat att arbeta med detta projekt har det också väckt ett intresse hos mig att skapa ett kompilerat språk. Jag skulle vilja försöka att skapa ett språk som kompilerar till *Assembly*-kod istället för att utvärderas med *Ruby*-kod. Det som skulle vara intressant med ett sådant projekt är att språket skulle kunna bli oberoende av andra språk och kunna skrivas i sig själv. Jag har ingen erfarenhet av att programmera i *Assembly* ännu och därför får detta projekt vara något för framtiden.