



ETL

Easy To Learn

TDP019 Projekt: Datorspråk

Språkdokumentation

Författare

Ahmed Sikh , ahmsi881@student.liu.se
Sayed Ismail Safwat, saysa289@student.liu.se

Innehåll

1	Revisionshistorik	2
2	Inledning	2
2.1	Syfte	2
2.2	Introduktion	2
2.3	Målgrupp	2
3	Användarhandledning	2
3.1	Installation	2
3.2	Variabler och Tilldelning	3
3.3	Matematiska Operationer	3
3.4	Kommentarer	3
3.5	Print	4
3.6	Villkor/If-satser	4
3.7	Iteration	6
3.8	Funktioner	6
3.9	Multiple Strings	8
4	Systemdokumentation	9
4.1	Lexikaliska Analys	9
4.2	Parsning	10
4.3	Kodstandard	10
5	Reflektion	11
6	Bilagor	12
6.1	BNF Grammatik	12
6.2	ETL.rb	14
6.3	classes.rb	18
6.4	etl.etl	23
7	Bilder	25

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första version av Språkdokumentation	210510

2 Inledning

Detta är ett projekt på IP-programmet som är skapat under den andra terminen vid Linköpings universitet i kursen TDP019 Projekt: datorspråk.

2.1 Syfte

Syftet med denna kursen var att visa vilka komponenter ett språk består av och hur ett nytt programmeringsspråk byggs upp med de där komponenterna.

2.2 Introduktion

I det här språket har tagits inspiration för det mesta från Ruby språket. ETL är utvecklats för en nybörjare användare och är skrivet i ett sätt som liknar skriftligt engelska vilket gör det möjligt för språkets läsbarhet.

2.3 Målgrupp

ETL (Easy To Learn) språket skall passa de nybörjare som har inga tidigare förkunskaper inom programmering. Det passar perfekt dem som vill börja lära sig programmering på rätt sätt som kommer täcka de mesta grunderna där en ny programmerare bör tänka på. Språket kommer även passa lärarna som vill lära ut programmering till de nybörjare eller möjligtvis till en grupp av barn i grundskolan.

3 Användarhandledning

3.1 Installation

För att kunna testa ETL krävs den senaste versionen av Ruby installerad.

För att kunna köra språket krävs det laddas ner. Språket kan laddas ner via länken:

<https://gitlab.liu.se/ahmsi881/tdp019/-/archive/master/tdp019-master.zip>

Användaren behöver skriva kommandoraden ***ruby ETL.rb*** för att kunna köra programmet.

Det finns två sätt att köra ETL språket på:

1. Att skriva kod genom terminalen, vilket är ett sätt om användaren vill skriva endast en enkel rad kod som inte består av flera saker samtidigt. Detta kan användaren göra i ETL.rb genom: se **Figur 1** i sektionen **Bilder!**.
2. Andra sättet är att testa språket i sin helhet vilket innebär att användaren skriver sin kod i en fil som heter **etl.etl** där kommer programmet ta hand om resten. Detta kan användaren göra i ETL.rb genom: se **Figur 2** i sektionen **Bilder!**.

3.2 Variabler och Tilldelning

Variabler har en dynamisk typning där användaren behöver inte specificera datatypen när den ska deklarerars. Tilldelningen i ETL betecknas endast med tilldelningsoperatoren “=”. I ETL går det att tilldela en variabel till booleska värden, strängar och matematiska uttryck.

Det innebär att det ska finnas endast ett namn och dennes värde vilket visas i följande stil:

```
x = 5
y = "Hej"
z = "hej" plus "då"
d = 5 < 10
```

3.3 Matematiska Operationer

ETL kan utföra alla sedvanliga matematiska beräkningar såsom addition, subtraktion, multiplikation, division, potenser och modulo samt deras rätta prioriteter och associativiteten det vill säga division och multiplikation ska utföras före addition och subtraktion. Samtliga beräkningar utförs oavsett de är heltal eller flyttal. Språket stöder även beräkningarna inuti en parentes.

Exempel:

```
(5 + 4)
1 - 5
2 * 1.0
5 / 5
4 - 7 * (10 / 2)
5 ^ 2
10 % 3
```

Det går även att utföra matematiska beräkningar på variabler som har heltal eller flyttal som värde. Ex:

```
x = 5
y = x + 2
z = x * y
```

3.4 Kommentarer

I språket finns det möjligheten att ignorera en rad eller flera rader ifall användaren inte vill att de raderna ska köras. Detta görs genom att skriva ”<comment>” för att ignorera en rad och för att ignorera fler rader måste det skrivas ”<comment>” i början av raden och ”<end>” i slutet av raden.

Exempel på flerradskommentar:

```
<comment
Detta är en flerradskommentar och allt som skrivs i det här utrymmet kommer ignoreras
och inte köras.
Som det syns här går det att skriva vad som helst. ?!"#$%&123456789
Det är jätteviktigt att inte glömma skriva <end i slutet av raden.
<end
```

Exempel på enkelradskommentar:

```
<< Här ignoreras bara en rad som skrevs med << i början av raden.
<< Varje rad måste ha << i början för att den ska ignoreras.
```

Kommenterar används ofta av programmerare som en påminnelse på hur dem har kommit fram till den specifika koden.

3.5 Print

I ETL går det att skriva ut datatyper som strängar, tal, logiska uttryck och flera strängar samtidigt förutsatt att de är tilldelade till en variabel innan utskriften. För att skriva ut används ordet **write** innan variabelnamnet. Exempel:

```
a = "Printing should be easy!"
write a
-----
b = 3 < 4
write b
-----
c = 1234
write c
```

Skriver ut följande:

```
-->> Printing 'Printing should be easy!'
-----
-->> Printing 'true'
-----
-->> Printing '1234'
```

3.6 Villkor/If-satser

Att skriva villkor eller if-satser i ETL språket är inte avancerad. Användaren bör börja med **“if”** i början av raden, sedan öppna en parentes där kan användaren skriva en eller flera logiska uttryck som kan ge **falsk** eller **sant**, efter det stänger användaren parentesen och skriver därefter ordet **“then”**. Då börjar användaren på en ny rad för att skriva den satsen eller de satserna som ska utföras ifall de logiska uttrycken som finns inuti parentesen ska returnera **sant**. I slutet av en if-sats ska användaren skriva **“endif”** för att säga att här slutar villkoret.

Exempel på en if-sats:

```
x = 7
y = 8
if (x > 6 and y == 8) then
write "if-sats fungerar"
endif
```

Skriver ut följande:

```
-->> Printing 'if-sats fungerar'
```

I ETL kan användaren skriva en elseif-sats som följer av en if sats. Detta kan användaren göra genom att skriva **“elseif”** i en ny rad. Det betyder att om de logiska uttrycken som finns inuti **if-sats** parentes returnerar **falsk**, så kommer **elseif-satsen** nu utföra den eller de satserna som finns efter ordet **“elseif”**.

Exempel på en elseif-sats:

```
j = 2
if (j != 2) then
write "if-sats fungerar"
elseif (j == 2) then
write "elseif-sats fungerar"
otherwise
write "otherwise fungerar"
endif
```

Skriver ut följande:

```
-->> Printing 'elseif-sats fungerar'
```

I ETL kan också användaren skriva en **else-sats** som följer av antingen en **if-sats** eller **elseif-sats**. Detta kan användaren göra genom att skriva **“otherwise”** i en ny rad. Det betyder att om de logiska uttrycken som finns inuti **if-sats** eller **elseif-sats** parenteser returnerar **falsk**, så kommer **else-satsen** nu utföra den eller de satser som finns efter ordet **“otherwise”**. I slutet användaren kommer också göra samma sak här det vill säga att skriva **“endif”** för att visa att här slutar villkoret.

Exempel på en if-sats som följer av en else-sats:

```
x = 7
y = 8
if (x less than 6 or y equal 9) then
write "if-sats fungerar"
otherwise
write "otherwise-sats fungerar"
endif
```

Skriver ut följande:

```
-->> Printing 'otherwise-sats fungerar'
```

I ETL kan användaren bestämma skriva logiska operator i tecken, exempelvis:

<, >, <=, >=, != och ==

eller att skriva logiska operator i ord, exempelvis:

less than, **greater than**, **less than or equal to**, **greater than or equal to**, **not equal to** och **equal**

ETL kan även hantera **or**, **and** och **not**, se exemplen ovan!

3.7 Iteration

Det finns en sort loop i ETL språket vilket kallas för en while-loop där användaren har möjlighet att iterera igenom exempelvis ett tal tills det villkoret i loopen har uppfyllts. För att skriva en while-loop skrivs först ordet **while** sedan villkoret inuti en parentes. Efter det går det att skriva den satsen eller de satserna när villkoret som finns inuti parentes uppfylls, därefter för att avsluta while-loopen måste ordet **endwhile** skrivas i en ny rad. Exempel:

```
y = 1
while ( y < 4)
write "while-loop fungerar"
y = y + 1
endwhile
```

Skriver ut följande:

```
-->> Printing 'while-loop fungerar'
-->> Printing 'while-loop fungerar'
-->> Printing 'while-loop fungerar'
```

I exemplet ovan, skrevs ut **'while-loop fungerar'** samt lägger till en etta till variabeln **y** så länge den uppfyller villkoret (**y < 4**). Detta innebär att satserna kommer utföras tills det variabeln **y** är mindre än fyra.

I ETL går det även att avbryta while-loopen genom att skriva **stop** efter de satserna som ska utföras för det första gången. Detta gör while-loopen att utföra de satserna endast en gång, därefter kommer det avbrytas.

Exempel:

```
y = 1
while ( y < 4)
write "while-loop fungerar endast en gång"
y = y + 1
stop
endwhile
```

Skriver ut följande:

```
-->> Printing 'while-loop fungerar endast en gång'
```

3.8 Funktioner

ETL språket har två sorts funktioner:

1. Funktioner utan parametrar:

För att definiera en funktion utan parameter i ETL behöver användaren skriva **define** sedan sätta ett namn på den funktionen. Därefter måste användaren skriva tom parentes så att kodraden kommer se ut så här: **define name()** i slutändan.

Efter definitionen av funktionen kan användaren skriva en eller flera satser inuti funktionskroppen. Funktionskroppen avslutas med ordet **return** beroende på vad användaren vill returnera.

I slutet bör alltid användaren skriva ordet **enddef** för att avsluta funktionskroppen.

Användaren kan anropa funktionen genom att skriva exempelvis **write name()** på en ny rad, där kommer terminalen skriva ut det som funktionen returnerar.

Exempel på funktioner utan parameter:

```
define add()
a = 4
b = 5
c = a + b
return c
enddef

write add()
```

Skriver ut följande:

```
-->> Function 'add' returning '9'
```

2. Funktioner med parametrar:

För att definiera en funktion med parametrar i ETL behöver användaren skriva **define** sedan sätta ett namn på den funktionen. Därefter måste användaren öppna en parentes för att skriva parameters namn. Funktionen kan ta flera parametrar som har kommatecken emellan. Kodraden kommer se ut så här: **define name(a ,b)** i slutändan.

Efter definitionen av funktionen kan användaren skriva en eller flera satser inuti funktionskroppen. Funktionskroppen avslutas med ordet **return** beroende på vad användaren vill returnera.

I slutet bör alltid användaren skriva ordet **enddef** för att avsluta funktionskroppen.

Användaren kan anropa funktionen med parametrar genom att skriva exempelvis **write name(2, 5)** på en ny rad, där parametrarna som finns inuti parentesen ska ta sina värde. I slutet kommer terminalen skriva ut det som funktionen returnerar.

Exempel på funktioner med parameter:

```
define add(a, b)
s = a + b
return s
enddef

write add(25, 75)
```

Skriver ut följande:

```
-->> Function 'add' returning '100'
```


3.9 Multiple Strings

ETL har en konstruktion som heter Multiple Strings. Denna finns för att låta användaren att addera antingen två eller flera variabler som innehåller strängar (som i Exempel 1) eller två eller flera strängar (som i Exempel 2) med varandra genom att skriva ordet **plus** mellan de variablerna/strängarna som ska adderas.

Exempel 1:

```
x = "ETL" plus " är enkelt."  
write x
```

Skriver ut följande:

```
-->> Printing 'ETL är enkelt.'
```

Exempel 2:

```
y = "ETL"  
z = " är"  
w = " lätt att lära sig!"  
write y plus z plus w
```

Skriver ut följande:

```
-->> Printing 'ETL är lätt att lära sig!'
```

4 Systemdokumentation

ETL språket uppbyggt på **rdparse.rb** som är tagen från båda TDP007 och TDP019 kurshemsidan. **rdparse.rb** hjälper med att göra den lexikaliska analysen samt själva parsning delen på den koden som användaren skriver.

ETL.rb och **classes.rb** filerna är skapade av oss senare under projektarbetet. Filen **classes.rb** består av alla noder som används i match reglerna i **ETL.rb** där alla reglerna som bestämmer syntaxen är skriven i.

4.1 Lexikaliska Analys

I lexikaliska analysen skapas de olika tokens som språket har i **ETL.rb**. Tokens består av reguljära uttryck(RegEx) som är en följd av flera tecken som matchar en viss mönster.

I slutet kommer alla tokens skickas vidare till parsen.

Här kommer alla tokens i samma ordning som de är på **ETL.rb** filen:

1. Tokens som inte ska parsas och kommer ignoreras:

- Matchar och ignorerar flerradskommentarer.

```
token(/\<comment[^\!]*\<end/)
```

- Matchar och ignorerar enkelradskommentar

```
token(/(<<.+$/)
```

- Matchar och ignorerar alla mellanrum

```
token(/\s+/)
```

2. Tokens som ska parsas:

- Matchar alla flyttal och returneras som Float”

```
token(/(\d+[\.]\d+)/) { |m| m.to_f }
```

- Matchar alla heltal och returneras som "Integer”

```
token(/\d+/) { |m| m.to_i }
```

- Matchar strängar inom enkelcitattecken

```
token(/'[^']*'/) { |m| m }
```

- Matchar strängar inom dubbelcitattecken

```
token(/"[^"]*" /) { |m| m }
```

- Matchar namn på variabler

```
token(/[a-z]+[a-z0-9_]* /) { |m| m }
```

- Matchar allt annat(enkla kårakterer)

```
token(/./) { |m| m }
```

4.2 Parsning

Efter att alla tokens har skickats från lexikaliska delen för parsning, och matchats de reglerna som beskriven i vår BNF-grammatiken då börjar parsern gör sitt jobb som är att hitta det mönstret från den koden som användaren skriver och bygga abstrakta syntaxträdet i slutet. Parsern körs rekursivt och går efter BNF-grammatiken.

Varje konstruktion i ETL språket har sin egen klass vilket varje klass har en **eval()** funktion som körs när programmet använder den relevanta klassen och dennes eval funktion.

Exempel: (Se **Figur 3** i sektionen **Bilder!**)

4.3 Kodstandard

Språket använder sig inte av något indentering vilket innebär att alla mellanrum kommer tas bort från koden som användaren skriver.

Vissa kodstandard som **ETL** har:

- I slutet av varje **if-sats** måste användaren avstänga kroppen genom att skriva **endif**.
- Efter varje if-statement måste användaren skriva **then**.
- I slutet av varje **while-loop** måste användaren avstänga kroppen genom att skriva **endwhile**.
- I slutet av varje **funktion** måste användaren avstänga kroppen genom att skriva **enddef**.
- Booleska uttryck kan skrivas antingen i numeriskt eller skriftligt sätt. Exempel: **<** eller **less than** osv.

5 Reflektion

I denna kursen var vi ombedd att skapa ett nytt programmeringsspråk och med våra kunskaper från tidigare kursen kändes det mycket svårt att tänka på hur och varifrån ska man börja med att skriva eller implementera. Det var lite svårt i början, eftersom man vet inte om man gör rätt eller fel osv, kanske för att man inte kunde testa allt man skriver precis som vi gjorde hittills i tidigare kurser där man kunde testa allt man vill under arbetet. Dock efter handledningstillfällen kom vi igång med vilket sort av språk vi kommer skapa då vi fick en bättre bild och kunde ta de första stegen. Att skriva all tokens och all BNF-grammatiken var relativt enkelt då kunde vi skriva dem klart mycket snabbare än vi trodde. Däremot var vi på fel spår och hade gått för långt med att skriva sakerna som inte var relevanta i den tidpunkten. Detta märkte vi tack vare vår handledare under en av handledningstillfällen som rekommenderade att vi borde ta saker ett steg i taget för att testa och se om de fungerar eller inte. Exempelvis man kan börja med matematik och operationer och sedan kan man börja med tilldelning och variabler och så vidare.

Under projektet var vi tvungna att ändra grammatiken ständigt eftersom vi ibland inte fick förväntade resultat så grammatiken förändrades till den bättre versionen hela tiden tills vi var klara.

Ett av de problemen, konstig nog, vi hade under arbetet var att minustecknet inte fungerade som det ska, dvs det fungerade bara när man skriver $(5 - 2)$ med mellanrum. Vi lyckades lösa problemet genom att ändra på vår tokens så att de matchar bara tal oavsett de är positiva eller negativa, sedan ändrade vi på Constant klassen så vi lade till en if-sats som säger om det är negativ så ska den siffran multipliceras med (-1) . Innan hade vi matchgrupp bara för Float och Integer i atom matchregel så vi behövde lägga till också de Float och Integer som behövs för negativa tal.

En annan sak som fick mer tid av oss var booleska uttryck hanteringen. Detta var viktigt för oss då vi behövde den för att gå vidare med att testa resten av programmen där ett boolesk uttryck används. Senare märkte vi att vi hade ingen matchgrupp för **'true'** och **'false'** för att känna till om något värde är falskt eller sant. Detta fick vi lösa genom att lägga till matchgrupp till **'false'** och **'true'** som också använder sig av klassen **Constant**. Då fick vi **or** och **and** fungera som det ska, men inte **not** eftersom **not** använde samma klass som **or/and** och det var inte så bra eftersom **or/and** klassen behöver ta in 3 arguments/parametrar men **not** behöver bara ha två, så vi behövde skapa klassen **Not** som kommer bara hantera det fallet för programmet.

Ett annat stort problem vi stött på under projektet var ordningen på **statement** matchgrupperna samt de andra matchgrupperna i BNF. Där vi började få samma felmeddelande för flera saker vi skapade. Detta tog lång tid för att hitta vart problemet är, där vi märkte i slutet att det ligger på ordningen där minst generella ska komma först i ordningen och mest generella ska vara i slutet. Det handlar mest om erfarenhet man får under projektarbetet, skulle vi vara medvetna på att minst generella ska vara först i ordningen så skulle det vara snabbt att fixa problemet eller kanske vi inte skulle hamna på detta problemet alls.

En av de svåraste delarna i språket var att skapa scopehanteringen vilket var på grund av att vi inte var säkra om det behövs i språket eller ej. Efter handledarens förklaring om scopehantering fick vi veta vad exakt scopehantering är och vilka saker man måste tänka för att implementera den. Vi fick veta att de är massa våningar för exempel våning 0 är det globala scopet och våning 1 är en lokal scope till exempel en funktion, där varje scope kommer ha sina egna variabler.

Om vi jämför språkspecifikations dokumentet med det slutliga arbetet så kan vi säga att vi har ändrat vår tanke med scopehanteringen, eftersom vi tycker att det är lättare för nybörjare att ha dynamisk istället för statisk scopehantering.

Avslutningsvis fick vi mycket stora erfarenheter som vi inte behärskade innan projektets gång och vi tycker också att vi har nått målet som var att förstå hur ett programmeringsspråk är uppbyggt samt vilka verktyg det behövs för att skapa ett eget programmeringsspråk.

6 Bilagor

6.1 BNF Grammatik

```

1 <PROGRAM>      ::= <STATEMENTS>
2
3 <STATEMENTS>   ::= <STATEMENTS> <STATEMENT>
4                  | <STATEMENT>
5
6 <STATEMENT>    ::= <RETURN>
7                  | <FUNC>
8                  | <FUNCCALL>
9                  | <STOP>
10                 | <PRINT>
11                 | <IF_BOX>
12                 | <WHILEITERATION>
13                 | <ASSIGN>
14
15 <ASSIGN>       ::= <ID> = <BOOL_LOGIC>
16                  | <ID> = <MULTIPLE_STRINGS>
17                  | <ID> = <STRING_EXPR>
18                  | <ID> = <EXPR>
19
20 <STRING_EXPR>  ::= /'[^']*'/
21                  | /"[^"]*" /
22
23 <MULTIPLE_STRINGS> ::= <STRING_EXPR> plus <STRING_EXPR>
24                  | <MULTIPLE_STRINGS> plus <STRING_EXPR>
25                  | <ID> plus <ID>
26                  | <MULTIPLE_STRINGS> plus <ID>
27
28 <EXPR>         ::= <EXPR> + <TERM>
29                  | <EXPR> - <TERM>
30                  | <TERM>
31
32 <TERM>         ::= <TERM> * <ATOM>
33                  | <TERM> / <ATOM>
34                  | <TERM> ^ <ATOM>
35                  | <TERM> % <ATOM>
36                  | <ATOM>
37
38 <BOOL_LOGIC>   ::= <BOOL_LOGIC> and <BOOL_LOGIC>
39                  | <BOOL_LOGIC> or <BOOL_LOGIC>
40                  | not <BOOL_LOGIC>
41                  | true
42                  | false
43                  | ( <BOOL_LOGIC> )
44                  | <BOOL_LIST>
45
46 <BOOL_LIST>    ::= <LESS_THAN>
47                  | <GREATER_THAN>
48                  | <LESS_THAN_OR_EQUAL_TO>
49                  | <GREATER_THAN_OR_EQUAL_TO>
50                  | <NOT_EQUAL_TO>
51                  | <EQUAL>
52
53 <LESS_THAN>    ::= <EXPR> < <EXPR>
54                  | <EXPR> less than <EXPR>
55
56 <GREATER_THAN> ::= <EXPR> > <EXPR>
57                  | <EXPR> greater than <EXPR>
58
59 <LESS_THAN_OR_EQUAL_TO> ::= <EXPR> <= <EXPR>
60                  | <EXPR> less than or equal to <EXPR>
61

```

```

62 <GREATER_THAN_OR_EQUAL_TO> ::= <EXPR> >= <EXPR>
63     | <EXPR> greater than or equal to <EXPR>
64
65 <NOT_EQUAL_TO> ::= <EXPR> != <EXPR>
66     | <EXPR> not equal to <EXPR>
67
68 <EQUAL>         ::= <EXPR> == <EXPR>
69     | <EXPR> equal <EXPR>
70
71 <ID>            ::= /[a-z]+[a-z0-9_]*/
72
73 <FUNC>          ::= define /[a-z]+[a-z0-9_]*/ ( <ARGUMENTS> ) <STATEMENTS> enddef
74     | define /[a-z]+[a-z0-9_]*/ ( ) <STATEMENTS> enddef
75
76 <FUNCCALL>      ::= <ID> ( )
77     | <ID> ( <ARGUMENT> )
78
79 <RETURN>        ::= return <ARGUMENT>
80
81 <ARGUMENTS>     ::= <ARGUMENTS> , <ARGUMENT>
82     | <ARGUMENT>
83
84 <ARGUMENT>      ::= <STRING_EXPR>
85     | <EXPR>
86
87 <WHILE_LOOP>    ::= while ( <BOOL_LOGIC> ) <STATEMENTS> endwhile
88
89 <STOP>          ::= stop
90
91 <IF_BOX>        ::= if ( <BOOL_LOGIC> ) then <STATEMENTS> endif
92     | if ( <BOOL_LOGIC> ) then <STATEMENTS> otherwise <STATEMENTS> endif
93     | if ( <BOOL_LOGIC> ) then <STATEMENTS> elseif ( <BOOL_LOGIC> ) then
94         <STATEMENTS> otherwise <STATEMENTS> endif
95
96 <PRINT>         ::= write <MULTIPLE_STRINGS>
97     | write <STRING_EXPR>
98     | write <BOOL_LOGIC>
99     | write <EXPR>
100
101 <ATOM>          ::= <FUNCTION_CALL>
102     | <Float>
103     | <Integer>
104     | - <Float>
105     | - <Integer>
106     | ( <EXPR> )
107     | <ID>

```



```

59     match(:multiple_strings, "plus", :id) { |mult_str, _, id| Plus_str.new("plus",
mult_str, id) }
60     end
61
62     rule :expr do
63       match(:expr, '+', :term) { |expr, _, term| Expr.new('+', expr, term) }
64       match(:expr, '-', :term) { |expr, _, term| Expr.new('-', expr, term) }
65       match(:term)
66     end
67
68     rule :term do
69       match(:term, '*', :atom) { |term, _, atom| Expr.new('*', term, atom) }
70       match(:term, '/', :atom) { |term, _, atom| Expr.new('/', term, atom) }
71       match(:term, '^', :atom) { |term, _, atom| Expr.new('^', term, atom) }
72       match(:term, '%', :atom) { |term, _, atom| Expr.new('%', term, atom) }
73       match(:atom)
74     end
75
76     rule :bool_logic do
77       match(:bool_logic, 'and', :bool_logic) { |lhs, _, rhs| Condition.new('and', lhs,
rhs) }
78       match(:bool_logic, 'or', :bool_logic) { |lhs, _, rhs| Condition.new('or', lhs, rhs) }
79       match('not', :bool_logic) { |_, oper| Not.new('not', oper) }
80       match('true') { Constant.new(true) }
81       match('false') { Constant.new(false) }
82       match('(', :bool_logic, ')') { |_, bool_log, _| bool_log }
83       match(:bool_list)
84     end
85
86     rule :bool_list do
87       match(:less_than)
88       match(:greater_than)
89       match(:less_than_or_equal_to)
90       match(:greater_than_or_equal_to)
91       match(:not_equal_to)
92       match(:equal)
93     end
94
95     rule :less_than do
96       match(:expr, '<', :expr) { |expr1, _, expr2| Condition.new('<', expr1, expr2) }
97       match(:expr, 'less', 'than', :expr) { |expr1, _, _, expr2| Condition.new('less
than', expr1, expr2) }
98     end
99
100    rule :greater_than do
101      match(:expr, '>', :expr) { |expr1, _, expr2| Condition.new('>', expr1, expr2) }
102      match(:expr, 'greater', 'than', :expr) { |expr1, _, _, expr2| Condition.new('greater
than', expr1, expr2) }
103    end
104
105    rule :less_than_or_equal_to do
106      match(:expr, '<', '=', :expr) { |expr1, _, _, expr2| Condition.new('<=', expr1,
expr2) }
107      match(:expr, 'less', 'than', 'or', 'equal', 'to', :expr) { |expr1, _, _, _, _,
expr2| Condition.new('less than or equal to', expr1, expr2) }
108    end
109
110    rule :greater_than_or_equal_to do
111      match(:expr, '>', '=', :expr) { |expr1, _, _, expr2| Condition.new('>=', expr1,
expr2) }
112      match(:expr, 'greater', 'than', 'or', 'equal', 'to', :expr) { |expr1, _, _, _, _,
expr2| Condition.new('greater than or equal to', expr1, expr2) }
113    end
114
115    rule :not_equal_to do

```



```

116     match(:expr, '!', '=', :expr) { |expr1, _, _, expr2| Condition.new('!=', expr1,
117     expr2) }
118     match(:expr, 'not', 'equal', 'to', :expr) { |expr1, _, _, _, expr2|
119     Condition.new('not equal to', expr1, expr2) }
120     end
121     rule :equal do
122     match(:expr, '=', '=', :expr) { |expr1, _, _, expr2| Condition.new('==', expr1,
123     expr2) }
124     match(:expr, 'equal', :expr) { |expr1, _, expr2| Condition.new('equal', expr1,
125     expr2) }
126     end
127     rule :id do
128     match(/[a-z]+[a-z0-9_]*/) { |id| Variable.new(id) }
129     end
130     rule :func do
131     match("define", /[a-z]+[a-z0-9_]*/, "(", :arguments, ")", :statements, "enddef") {
132     |_, def_name, _, args, _, states, _|
133     Function.new(def_name, args, states) }
134     match("define", /[a-z]+[a-z0-9_]*/, "(", ")", :statements, "enddef") { |_, def_name,
135     _, _, states, _| Function.new(def_name, Array.new, states) }
136     end
137     rule :funcCall do
138     match(:id, "(", " ") { |def_name, _, _| FunctionCall.new(def_name, Array.new) }
139     match(:id, "(", :arguments, " ") { |def_name, _, args, _| FunctionCall.new(def_name,
140     args) }
141     end
142     rule :return do
143     match("return", :argument) { |_, arg| Return.new(arg) }
144     end
145     rule :arguments do
146     match(:arguments, ',', :argument){ |args, _, arg| [args, arg].flatten }
147     match(:argument)
148     end
149     rule :argument do
150     match(:string_expr)
151     match(:expr)
152     end
153     rule :whileIteration do
154     match("while", "(", :bool_logic, ")", :statements, "endwhile") { |_, _, bool_log, _,
155     states, _| While.new(bool_log, states) }
156     end
157     rule :stop do
158     match("stop") { |_| Stop.new() }
159     end
160     rule :if_box do
161     match("if", "(", :bool_logic, ")", "then", :statements, "endif") { |_, _, bool_log,
162     _, _, if_states, _| If.new(bool_log, if_states) }
163     match("if", "(", :bool_logic, ")", "then", :statements, "otherwise", :statements,
164     "endif") { |_, _, bool_log, _, _, if_states, _, else_states, _|
165     If.new(bool_log, if_states, else_states) }
166     match("if", "(", :bool_logic, ")", "then", :statements, "elseif", "(", :bool_logic,
167     ")", "then", :statements, "otherwise", :statements, "endif") { |_, _, bool_log, _, _,
168     if_states, _, _, elsif_bool_log, _, _, elsif_state, _, else_states, _|
169     If.new(bool_log, if_states, elsif_bool_log, elsif_state, else_states) }
170     end

```

```

169
170     rule :print do
171         match("write", :multiple_strings) { |_, mult_str| Print.new(mult_str) }
172         match("write", :string_expr) { |_, str_exp| Print.new(str_exp) }
173         match("write", :bool_logic) { |_, bool_log| Print.new(bool_log) }
174         match("write", :expr) { |_, exp| Print.new(exp) }
175     end
176
177     rule :atom do
178         match(:funcCall)
179         match(Float) { |float_num| Constant.new(float_num) }
180         match(Integer) { |int_num| Constant.new(int_num) }
181         match("-", Float) { |a, b| Constant.new(b, a) }
182         match("-", Integer) { |a, b| Constant.new(b, a) }
183         match('(', :expr, ')') { |_, exp, _| Expression.new(exp) }
184         match(:id)
185     end
186
187     end #end för all grammatik
188     #+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ END BNF +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
189     end #end för initialize
190
191     def done(str)
192         ["quit", "exit", "bye", "close", "stop"].include?(str.chomp)
193     end
194
195     #För att starta programmet i terminalen
196     def activate_terminal
197         print "[ETL] "
198         str = gets
199         if done(str) then
200             puts "Bye."
201         else
202             parsePrinter = @etlParser.parse str
203             puts "=> #{parsePrinter.eval}"
204             activate_terminal
205         end
206     end
207
208     #För att testa från en fil
209     def activate_file(etl_file)
210         @output = []
211         etl_file = File.read(etl_file)
212         @output = @etlParser.parse(etl_file)
213         @output
214     end
215
216     def log(state = true)
217         if state
218             @etlParser.logger.level = Logger::DEBUG
219         else
220             @etlParser.logger.level = Logger::WARN
221         end
222     end
223
224     end #end för klassen
225
226     checkEtl = Etl.new
227     checkEtl.log(false)
228     #checkEtl.activate_terminal
229     checkEtl.activate_file("etl.etl")
230     checkEtl.output.each { |segment|
231         if segment.class != Function and segment.class != FunctionCall
232             segment.eval()
233         end }

```

6.3 classes.rb

```
1 ## Alla klasser som behövs
2
3 $our_funcs = Hash.new
4
5 class ScopeHandler
6   def initialize()
7     @@level = 1
8     @@holder = {}
9   end
10  def defineScope(s)
11    @@holder = s
12    return @@holder
13  end
14  def receiveHolder()
15    return @@holder
16  end
17  def receiveLevel()
18    return @@level
19  end
20  def incre()
21    @@level = @@level + 1
22    return @@holder
23  end
24  def decre(s)
25    defineScope(s)
26    @@level = @@level - 1
27    return nil
28  end
29 end
30
31 $scope = ScopeHandler.new
32
33 def look_up(variable, our_vars)
34   levelNr = $scope.receiveLevel
35   if our_vars == $scope.receiveHolder
36     loop do
37       if our_vars[levelNr] != nil and our_vars[levelNr][variable] != nil
38         return our_vars[levelNr][variable]
39       end
40       levelNr = levelNr - 1
41       break if (levelNr < 0)
42     end
43
44     if our_vars[levelNr] == nil
45       our_vars[variable]
46     end
47   end
48 end
49
50 class Variable
51   attr_accessor :variable_name
52   def initialize(id)
53     @variable_name = id
54   end
55   def eval
56     return look_up(@variable_name, $scope.receiveHolder)
57   end
58 end
59
60 class Expr
61   attr_accessor :sign, :lhs, :rhs
62   def initialize(sign, lhs, rhs)
63     @sign = sign
```

```

64     @lhs = lhs
65     @rhs = rhs
66   end
67   def eval()
68     case sign
69       when '+'
70         return lhs.eval + rhs.eval
71       when '-'
72         return lhs.eval - rhs.eval
73       when '*'
74         return lhs.eval * rhs.eval
75       when '/'
76         return lhs.eval / rhs.eval
77       when '^'
78         return lhs.eval ** rhs.eval
79       when '%'
80         return lhs.eval % rhs.eval
81       else nil
82     end
83   end
84 end
85
86 class Plus_str
87   attr_accessor :sign, :lhs, :rhs
88   def initialize(sign, lhs, rhs)
89     @sign = sign
90     @lhs = lhs
91     @rhs = rhs
92   end
93   def eval()
94     case @sign
95       when 'plus'
96         return @lhs.eval + @rhs.eval
97       else nil
98     end
99   end
100 end
101
102 class Condition
103   attr_accessor :sign, :lhs, :rhs
104   def initialize(sign, lhs, rhs)
105     @sign = sign
106     @lhs = lhs
107     @rhs = rhs
108   end
109   def eval()
110     case sign
111       when '<', 'less than'
112         return lhs.eval < rhs.eval
113       when '>', 'greater than'
114         return lhs.eval > rhs.eval
115       when '<=', 'less than or equal to'
116         return lhs.eval <= rhs.eval
117       when '>=', 'greater than or equal to'
118         return lhs.eval >= rhs.eval
119       when '!=', 'not equal to'
120         return lhs.eval != rhs.eval
121       when '==', 'equal'
122         return lhs.eval == rhs.eval
123       when 'and'
124         return lhs.eval && rhs.eval
125       when 'or'
126         return lhs.eval || rhs.eval
127       else nil
128     end

```

```
129     end
130 end
131
132 class Not
133   attr_accessor :sign, :oper
134   def initialize(sign, oper)
135     @sign = sign
136     @oper = oper
137   end
138   def eval()
139     case sign
140     when 'not'
141       return (not oper.eval)
142     else nil
143     end
144   end
145 end
146
147 class Expression
148   def initialize(value)
149     @value = value
150   end
151   def eval()
152     @value.eval
153   end
154 end
155
156 class Assign
157   attr_reader :variable, :assign_expr
158   def initialize(variable, assign_expr)
159     @variable = variable
160     @assign_expr = assign_expr
161   end
162   def eval
163     value = @assign_expr.eval
164     @level_Nr = $scope.receiveLevel
165     scp = $scope.receiveHolder
166     if scp[@level_Nr] != nil
167       if scp[@level_Nr].has_key?(@variable.variable_name)
168         return scp[@level_Nr][@variable.variable_name] = value
169       else
170         scp[@level_Nr][@variable.variable_name] = value
171         return $scope.defineScope(scp)
172       end
173     elsif scp[@level_Nr] = {} and scp[@level_Nr][@variable.variable_name] = value
174       return $scope.defineScope(scp)
175     end
176   end
177 end
178
179 class Constant
180   attr_accessor :value
181   def initialize (value, negative = nil)
182     @value = value
183     @negative = negative
184   end
185   def eval()
186     if @negative
187       @value * -1
188     else
189       @value
190     end
191   end
192 end
193
```

```
194 class Print
195   def initialize(value)
196     @value = value
197   end
198   def eval()
199     #puts
200     if @value.eval != nil
201       puts "-->> Printing '#{@value.eval}'"
202       @value.eval
203     else
204       nil
205     end
206   end
207 end
208
209 class If
210   attr_accessor :if_bool_logic, :states, :elsif_bool_logic, :elsif_state, :otherwise_states
211   def initialize(if_bool_logic, states, elsif_bool_logic = nil, elsif_state = nil,
212     otherwise_states = nil)
213     @if_bool_logic = if_bool_logic
214     @states = states
215     @elsif_bool_logic = elsif_bool_logic
216     @elsif_state = elsif_state
217     @otherwise_states = otherwise_states
218   end
219   def eval()
220     if @if_bool_logic.eval()
221       @states.eval()
222     elsif @elsif_bool_logic.eval()
223       @elsif_state.eval()
224     elsif @otherwise_states != nil
225       @otherwise_states.eval()
226     end
227   end
228 end
229
230 class While
231   attr_accessor :bool_logic, :states
232   def initialize(bool_logic, states)
233     @bool_logic = bool_logic
234     @states = states
235   end
236   def eval()
237     check_stop = false
238     while @bool_logic.eval
239       @states.each { |segment|
240         if (segment.eval() == "stop")
241           check_stop = true
242         end }
243       if (check_stop == true)
244         break
245       end
246     end
247     @states
248   end
249 end
250
251 class Stop
252   def initialize()
253   end
254   def eval()
255     return "stop"
256   end
257 end
```

```

258
259 class Function
260   attr_accessor :def_name, :f_arguments, :states
261   def initialize(def_name, f_arguments, states)
262     @def_name = def_name
263     @f_arguments = f_arguments
264     @states = states
265     if !$our_funcs.has_key?(@def_name)
266       $our_funcs[@def_name] = self
267     else
268       raise("OOOPS! THE FUNCTION \"#{@def_name}\" DOES ALREADY EXIST!")
269     end
270   end
271   def recieveStates()
272     @states
273   end
274   def recieveArgs()
275     @f_arguments
276   end
277 end
278
279 class FunctionCall
280   attr_accessor :def_name, :f_c_arguments
281   def initialize(def_name, f_c_arguments)
282     @def_name = def_name
283     @f_c_arguments = f_c_arguments
284     @states = $our_funcs[@def_name.variable_name].recieveStates
285     @f_arguments = $our_funcs[@def_name.variable_name].recieveArgs
286
287     if !$our_funcs.has_key?(@def_name.variable_name)
288       raise("OOOPS! THERE IS NO FUNCTION CALLED '#{@def_name.variable_name}' ")
289     end
290     if (@f_c_arguments.length != @f_arguments.length)
291       raise("FAIL! WRONG NUMBER OF ARGUMENTS. (GIVEN #{@f_c_arguments.length} EXPECTED
292         #{@f_arguments.length})")
293     end
294   end
295   def eval()
296     scp = $scope.incre
297     funcArgs_len = 0
298     funcCallArgs_len = @f_c_arguments.length
299     while (funcArgs_len < funcCallArgs_len)
300       scp[@f_arguments[funcArgs_len].variable_name] = @f_c_arguments[funcArgs_len].eval
301       funcArgs_len = funcArgs_len + 1
302     end
303     @states.each { |state|
304       if state.class == Return
305         puts "-->> Function '#{@def_name.variable_name}' returning '#{state.eval}'"
306         break
307       else
308         state.eval
309       end }
310     scp.delete($scope.receiveLevel)
311     $scope.decre(scp)
312   end
313 end
314
315 class Return
316   def initialize(value)
317     @value = value
318   end
319   def eval
320     return @value.eval
321   end
322 end

```

6.4 etl.etl

```

1 write "*****"
2 write "***      Matematik Test      ***"
3 write "*****"
4
5 write 2 + 4
6 write 2 - 4
7 write 4 - 3
8 write 2 * 4 + 1
9 num = (2 * 4) + 1
10 write num
11 write 4 / 2 * 5
12 write 2 ^ 10
13 write 4 % 3
14
15
16 write "*****"
17 write "***      While-loop Test      ***"
18 write "*****"
19
20 y = 1
21 while (y < 5)
22 write "while loop fungerar"
23 <<stop
24 y = y + 1
25 endwhile
26
27 write "*****"
28 write "***      If-sats Test      ***"
29 write "*****"
30
31 x = 7
32 u = 8
33 if (x > 6 and u equal 8) then
34 write "if-sats fungerar"
35 otherwise
36 write "otherwise fungerar"
37 endif
38
39 write "*****"
40 write "***      Elseif-sats Test      ***"
41 write "*****"
42
43 i = 2
44 h = 10
45 if (i != 2) then
46 write "if-sats fungerar"
47 elseif (h == 10) then
48 write "elseif-sats fungerar"
49 otherwise
50 write "otherwise fungerar"
51 endif
52
53 write "*****"
54 write "***      Otherwise-sats Test      ***"
55 write "*****"
56
57 r = 2
58 t = 10
59 if (r != 2) then
60 write "if-sats fungerar"
61 elseif (t != 10) then
62 write "elseif-sats fungerar"
63 otherwise

```



```
64 write "otherwise fungerar"
65 endif
66
67
68 write "*****"
69 write "*** Funktioner utan parameter Test ***"
70 write "*****"
71
72 c = 10
73
74 define add()
75 a = 4
76 b = 5
77 w = 99999999999
78 c = a + b
79 return c
80 enddef
81 write add()
82
83 write c
84 write w
85
86
87 write "*****"
88 write "*** Funktioner med parameter Test ***"
89 write "*****"
90
91 s = 2
92
93 define foo(w, k)
94 s = w + k
95 j = 456456456456
96 return s
97 enddef
98 write foo(25, 75)
99
100 write s
101 write j
102
103 write "*****"
104 write "*** Multiple strings Test ***"
105 write "*****"
106
107 n = "Ahmed Sikh"
108 b = " Ismail"
109 v = " !"
110
111 write "Hej" plus " på dig"
112
113 write "ETL" plus " är" plus " lätt"
114
115 write b plus v
116
117 write n plus v plus b
```

7 Bilder

Figur 1: Exempel på hur ska det se ut när användaren vill testa språket genom terminalen

```
238 checkEtl = Etl.new
239 checkEtl.log(false)
240 | checkEtl.activate_terminal
241 | #checkEtl.activate_file("etl.etl")
242 | checkEtl.output.each { |segment|
243 |   | if segment.class != Function and segment.class != FunctionCall
244 |   |   | segment.eval()
245 |   | end }
246
247 |
```

Figur 2: Exempel på hur ska det se ut när användaren vill testa språket genom en test fil

```
238 checkEtl = Etl.new
239 checkEtl.log(false)
240 | #checkEtl.activate_terminal
241 | checkEtl.activate_file("etl.etl")
242 | checkEtl.output.each { |segment|
243 |   | if segment.class != Function and segment.class != FunctionCall
244 |   |   | segment.eval()
245 |   | end }
246
```

Figur 3: Här parsas en konstruktion där flera strängar adderas med varandra med hjälp av klass objektet som skapas av klassen **Plus_str**.

