



ETL

Easy To Learn

TDP019 Projekt: Datorspråk

Språkdokumentation

Författare

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

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första version av Språkdokumentation	210510
1.1	Andra version av Språkdokumentation	210522

Innehåll

1	Inledning	2
1.1	Syfte	2
1.2	Målgrupp	2
2	Användarhandledning	3
2.1	Installation	3
2.2	Variabler och Tilldelning	4
2.3	Matematiska Operationer	4
2.4	Kommentarer	4
2.5	Print	5
2.6	Villkor/If-satser	5
2.7	Iteration	7
2.8	Funktioner	7
2.9	Multiple Strings	9
3	Systemdokumentation	10
3.1	Lexikaliska Analys	10
3.2	Parsning	11
3.3	Kodstandard	12
4	Reflektion	13
5	Bilagor	14
5.1	BNF Grammatik	14
5.2	ETL.rb	16
5.3	classes.rb	20
5.4	etl.etl	25

1 Inledning

Detta är ett projekt på IP-programmet som är skapat under den andra terminen vid Linköpings universitet i kursen TDP019 Projekt: datorspråk. ETL (Easy To Learn) har inspirerats för det mesta från Ruby språket. Det har utvecklats för en nybörjare och är skrivet på ett sätt som liknar skriftligt engelska vilket ökar språkets läsbarhet.

1.1 Syfte

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

1.2 Målgrupp

ETL språket skall passa de nybörjare som inte har några förkunskaper inom programmering. Det passar perfekt dem som vill börja lära sig programmering på rätt sätt eftersom det täcker de elementära grunderna i programmering. Språket kommer även passa de lärare som vill lära ut programmering till nybörjare eller möjligtvis till en grupp av barn i grundskolan.

2 Användarhandledning

Den här sektionen innehåller instruktioner för att hur man ska komma igång med språket samt lära känna språkkonstruktioner.

2.1 Installation

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

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

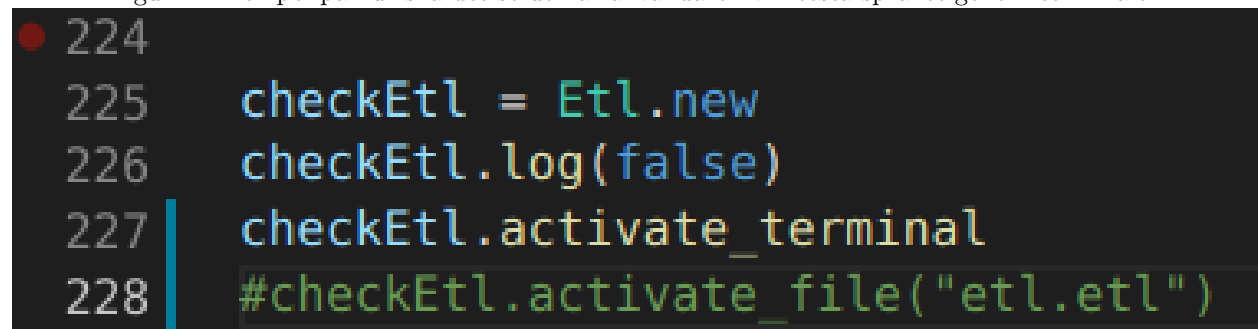
<https://gitlab.liu.se/ahmsi881/tdp019/-/archive/master/tdp019-master.zip>

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**.

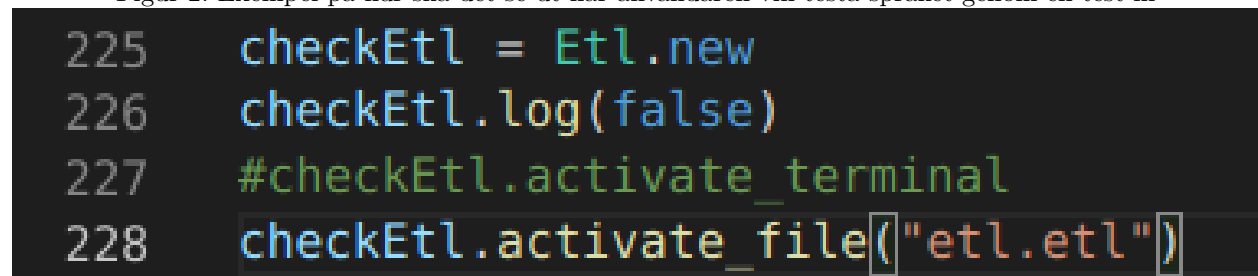
Figur 1: Exempel på hur ska det se ut när användaren vill testa språket genom terminalen



```
224
225   checkEtl = Etl.new
226   checkEtl.log(false)
227   checkEtl.activate_terminal
228   #checkEtl.activate_file("etl.etl")
```

2. Andra sättet är att testa språket i sin helhet vilket innebär att användaren skriver sin kod i en fil som heter **etl.etl** och där kommer programmet ta hand om resten. Detta kan användaren göra i ETL.rb genom: se **Figur 2**.

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



```
225   checkEtl = Etl.new
226   checkEtl.log(false)
227   #checkEtl.activate_terminal
228   checkEtl.activate_file("etl.etl")
```

2.2 Variabler och Tilldelning

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

2.3 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 om 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.

Exempel:

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

2.4 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 ”<<” 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 om hur de har kommit fram till den specifika koden.

2.5 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'
```

2.6 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 användaren kan skriva en eller flera logiska uttryck som kan ge **falskt** 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 följs 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-parentesen** returnerar **falsk**, så kommer **else-satsen** nu utföra den eller de satser som finns efter ordet **“otherwise”**. I slutet kommer användaren också att 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 skriva logiska operator i tecken.

Exempel 1:

<, >, <=, >=, != 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 exempel 1.

2.7 Iteration

Det finns en sorts loop i ETL-språket som 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 **'while-loop fungerar'** ut samt lägger till en etta till variabeln **y** så länge den uppfyller villkoret (**y < 4**). Detta innebär att satserna kommer att utföras tills 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 första gången. Detta gör while-loopen 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'
```

2.8 Funktioner

ETL-språket har två sorts funktioner:

1. Funktioner utan parametrar:

För att definiera en funktion utan parameter i ETL behöver användaren skriva **define** och sedan sätta ett namn på den funktionen. Därefter måste användaren skriva tom parentes så att kodraden kommer att 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 namnet. Funktionen kan ta flera parametrar som har kommatecken emellan. Kodraden kommer att 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 värden. 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'
```

2.9 Multiple Strings

ETL har en konstruktion som heter *Multiple Strings*. Denna finns för att låta användaren 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!'
```

3 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.

3.1 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[\w\W\s]*<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 käraktarer)

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

3.2 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**.

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



3.3 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.

4 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.

5 Bilagor

5.1 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 <WHILEITERATION> ::= 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>

```



```

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

```

```

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

```

19 / 26

5.3 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 $scope = ScopeHandler.new
31
32 def look_up(variable, our_vars)
33   levelNr = $scope.receiveLevel
34   if our_vars == $scope.receiveHolder
35     loop do
36       if our_vars[levelNr] and our_vars[levelNr][variable]
37         return our_vars[levelNr][variable]
38       end
39       levelNr = levelNr - 1
40       break if (levelNr < 0)
41     end
42
43     if our_vars[levelNr] == nil
44       our_vars[variable]
45     end
46   end
47 end
48
49 class Variable
50   attr_accessor :variable_name
51   def initialize(id)
52     @variable_name = id
53   end
54   def eval
55     return look_up(@variable_name, $scope.receiveHolder)
56   end
57 end
58
59 class Expr
60   attr_accessor :sign, :lhs, :rhs
61   def initialize(sign, lhs, rhs)
62     @sign = sign
63     @lhs = lhs
```

```

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

```

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

```

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

```



```

258 attr_accessor :def_name, :f_arguments, :states
259 def initialize(def_name, f_arguments, states)
260   @def_name = def_name
261   @f_arguments = f_arguments
262   @states = states
263   if !$our_funcs.has_key?(@def_name)
264     $our_funcs[@def_name] = self
265   else
266     raise("OOOPS! THE FUNCTION \"#{@def_name}\" DOES ALREADY EXIST!")
267   end
268 end
269 def recieveStates()
270   @states
271 end
272 def recieveArgs()
273   @f_arguments
274 end
275 def eval()
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 = $scope.incre
297     funcArgs_len = 0
298     funcCallArgs_len = @f_c_arguments.length
299     while (funcArgs_len < funcCallArgs_len)
300       scp[@f_arguments[funcArgs_len].variable_name] = @f_c_arguments[funcArgs_len].eval
301       funcArgs_len = funcArgs_len + 1
302     end
303     @states.each { |state|
304       if state.class == Return
305         puts "-->> Function '#{def_name.variable_name}' returning '#{state.eval}'"
306       else
307         state.eval
308       end
309     }
310     scp.delete($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

```

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