LINKÖPINGS UNIVERSITET

ETL

Easy To Learn

TDP019 Projekt: Datorspråk

Språkdokumentation

Författare

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

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2021
Version 1.0

6 maj 2021

Innehåll

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första version av Språkdokumentation	210505

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 deklareras. Tilldelningen i ETL betecknas endast med tilldelningsoperatorn "=". 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 och division 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. Ex:

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

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 **"< <"** 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 variabel namnet. 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 är Sant"
endif
```

Skriver ut följande:

```
-->> Printing 'if-sats är Sant'
```

I ETL kan också användaren skriva en else-sats som följer efter en if sats. Detta kan användaren göra genom att skriva **"otherwise"** i en ny rad. Det betyder att om de logiska uttrycken som finns inuti parentes returnerar **falsk**, så kommer else satsen nu utföra den eller de satser som finns efter ordet **"otherwise"**. I slutet användare 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 är Sant"
otherwise
write "otherwise-sats är Sant"
endif
```

Skriver ut följande:

```
-->> Printing 'otherwise-sats är Sant'
```

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 töm 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 två eller flera strängar med varandra genom att skriva ordet **plus** mellan de strängarna som ska adderas. Exempel:

```
x = "ETL" plus " är enkelt."
write x
-------------------------------------
y = "ETL"
z = " är lätt"
write y plus z plus " att lära sig."
```

Skriver ut följande:

```
-->> Printing 'ETL är enkelt.'
-----------------------------------------
-->> 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 dubbeltcitattecken
     ```
     token(/"[^\"]*"/) { |m| m }
     ```
   - Matchar namn på variabler
     ```
     token(/[a-z]+[a-z0-9_]*/) { |m| m }
     ```
   - Matchar allt annat(enkla käraktarer)
     ```
     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 det 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 variablar.

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

```
<PROGRAM>     ::= <STATEMENTS>

<STATEMENTS>  ::= <STATEMENTS> <STATEMENT>
                | <STATEMENT>

<STATEMENT>   ::= <RETURN>
                | <FUNC>
                | <FUNCCALL>
                | <STOP>
                | <PRINT>
                | <IF_BOX
                | <WHILEITERATION>
                | <ASSIGN>

<ASSIGN>      ::= <ID> = <BOOL_LOGIC>
                | <ID = <MULTIPLE_STRINGS>
                | <ID> = <STRING_EXPR>
                | <ID> = <EXPR>

<STRING_EXPR> ::= /'[^\']*'/
                | /"[^\"]*"/

<MULTIPLE_STRINGS> ::= <STRING_EXPR> plus <STRING_EXPR>
                | < MULTIPLE_STRINGS> plus <STRING_EXPR>
                | <ID> plus <ID>
                | <MULTIPLE_STRINGS> plus <ID>

<EXPR>        ::= <EXPR> + <TERM>
                | <EXPR> - <TERM>
                | <TERM>

<TERM>        ::= <TERM> * <ATOM>
                | <TERM> / <ATOM>
                | <ATOM>

<BOOL_LOGIC>  ::= <BOOL_LOGIC> and <BOOL_LOGIC>
                | <BOOL_LOGIC> or <BOOL_LOGIC>
                | not <BOOL_LOGIC>
                | true
                | false
                | ( <BOOL_LOGIC> )
                | <BOOL_LIST>

<BOOL_LIST>   ::= <LESS_THAN>
                | <GREATER_THAN>
                | <LESS_THAN_OR_EQUAL_TO>
                | <GREATER_THAN_OR_EQUAL_TO>
                | <NOT_EQUAL_TO>
                | <EQUAL>

<LESS_THAN>   ::= <EXPR> < <EXPR>
                | <EXPR> less than <EXPR>

<GREATER_THAN> ::= <EXPR> > <EXPR>
                | <EXPR> greater than <EXPR>

<LESS_THAN_OR_EQUAL_TO> ::= <EXPR> <= <EXPR>
                | <EXPR> less than or equal to <EXPR>

<GREATER_THAN_OR_EQUAL_TO> ::= <EXPR> >= <EXPR>
                | <EXPR> greater than or equal to <EXPR>
```

```
62
63 <NOT_EQUAL_TO>::= <EXPR> != <EXPR>
64                  | <EXPR> not equal to <EXPR>
65
66 <EQUAL>        ::= <EXPR> == <EXPR>
67                  | <EXPR> equal <EXPR>
68
69 <ID>           ::= /[a-z]+[a-z0-9_]*/
70
71 <FUNC>         ::= define /[a-z]+[a-z0-9_]*/ ( <ARGUMENTS> ) <STATEMENTS> enddef
72                  | define /[a-z]+[a-z0-9_]*/ ( ) <STATEMENTS> enddef
73
74 <FUNCCALL>     ::= <ID> ( )
75                  | <ID> ( <ARGUMENT> )
76
77 <RETURN>       ::= return <ARGUMENT>
78
79 <ARGUMENTS>    ::= <ARGUMENTS> , <ARGUMENT>
80                  | <ARGUMENT>
81
82 <ARGUMENT>     ::= <MULTIPLE_STRINGS>
83                  | <STRING_EXPR>
84                  | <EXPR>
85                  | <BOOL_LOGIC>
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
94 <PRINT>        ::= write <MULTIPLE_STRINGS>
95                  | write <STRING_EXPR>
96                  | write <BOOL_LOGIC>
97                  | write <EXPR>
98
99 <ATOM>         ::= <FUNCTION_CALL>
100                 | <Float>
101                 | <Integer>
102                 | - <Float>
103                 | - <Integer>
104                 | ( <EXPR> )
105                 | <ID>
```

6.2 ETL.rb

```
##Alla tokens, matchregler och matchgrupper

require './rdparse.rb'
require './classes.rb'

class Etl
    attr_accessor :output
    def initialize
        @etlParser = Parser.new("ETL") do
      #+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- BEGIN TOKENS +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
        token(/\<comment[^!]*\<end/) #parsa inte och ignorera flerarads kommentarer
        token(/(<<.+$)/) #parsa inte och ignorera en rad kommentar
        token(/\s+/)  #mellanrum ska inte parsas och ignoreras
        token(/(\d+[.]\d+)/) { |m| m.to_f } #floattal
        token(/\d+/) { |m| m.to_i } #heltal
        token(/'[^\']*'/) { |m| m } #sträng inom enkeltcitattecken (' ')
        token(/"[^\"]*"/) { |m| m } #sträng inom dubbeltcitattecken (" ")
        token(/[a-z]+[a-z0-9_]*/) { |m| m } #namn på variabler
        token(/./) { |m| m } #allt annat(enkla käraktarer)
      #+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- END TOKENS +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

      #+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- BEGIN BNF +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        start :program do
            match(:statements)
            end

        rule :statements do
            match(:statements,:statement){|states, state| [states, state].flatten}
            match(:statement)
        end

        rule :statement do
            match(:return)
            match(:func)
            match(:funcCall)
            match(:stop)
            match(:print)
            ##match(:bool_logic)
            match(:if_box)
            match(:whileIteration)
            match(:assign)
            ##match(:multiple_strings)
            ##match(:string_expr)
            ##match(:expr)
            end

        rule :assign do
            match(:id, "=", :bool_logic) { |variable_name, _, bool_log|
    Assign.new(variable_name, bool_log) }
            match(:id, "=", :multiple_strings) { |variable_name, _, mult_str|
    Assign.new(variable_name, mult_str) }
            match(:id, "=", :string_expr) { |variable_name, _, str_exp|
    Assign.new(variable_name, str_exp) }
            match(:id, "=", :expr) { |variable_name, _, expr| Assign.new(variable_name, expr) }
            end

        rule :string_expr do
            match(/'[^\']*'/) { |string| str = Constant.new(string[1, string.length-2]) }
            match(/"[^\"]*"/) { |string| str = Constant.new(string[1, string.length-2]) }
            #match(:funcCall)
            end

```

```
61          rule :multiple_strings do
62              match(:string_expr, "plus", :string_expr) { |str_exp1, _, str_exp2|
    Plus_str.new("plus", str_exp1, str_exp2) }
63              match(:multiple_strings, "plus", :string_expr) { |mult_str, _, str_exp|
    Plus_str.new("plus", mult_str, str_exp) }
64              match(:id, "plus", :id) { |id1, _, id2| Plus_str.new("plus", id1, id2) }
65              match(:multiple_strings, "plus", :id) { |mult_str, _, id| Plus_str.new("plus",
    mult_str, id) }
66              end
67
68          rule :expr do
69              match(:expr, '+', :term) { |expr, _, term| Expr.new('+', expr, term) }
70              match(:expr, '-', :term) { |expr, _, term| Expr.new('-', expr, term) }
71              match(:term)
72              end
73
74          rule :term do
75              match(:term, '*', :atom) { |term, _, atom| Expr.new('*', term, atom) }
76              match(:term, '/', :atom) { |term, _, atom| Expr.new('/', term, atom) }
77              #match(:funcCall)
78              match(:atom)
79              end
80
81         rule :bool_logic do
82              match(:bool_logic, 'and', :bool_logic) { |lhs, _, rhs| Condition.new('and', lhs,
    rhs) }
83              match(:bool_logic, 'or', :bool_logic) { |lhs, _, rhs| Condition.new('or', lhs, rhs) }
84              match('not', :bool_logic) { |_, oper| Not.new('not', oper) }
85              match('true') { Constant.new(true) }
86              match('false') { Constant.new(false) }
87              match('(', :bool_logic, ')') { |_, bool_log, _| bool_log }
88              match(:bool_list)
89              end
90
91          rule :bool_list do
92              match(:less_than)
93              match(:greater_than)
94              match(:less_than_or_equal_to)
95              match(:greater_than_or_equal_to)
96              match(:not_equal_to)
97              match(:equal)
98          end
99
100         rule :less_than do
101             match(:expr, '<', :expr) { |expr1, _, expr2| Condition.new('<', expr1, expr2) }
102             match(:expr, 'less', 'than', :expr) { |expr1, _, _, expr2| Condition.new('less
    than', expr1, expr2) }
103         end
104
105         rule :greater_than do
106             match(:expr, '>', :expr) { |expr1, _, expr2| Condition.new('>', expr1, expr2) }
107             match(:expr, 'greater', 'than', :expr) { |expr1, _, _, expr2| Condition.new('greater
    than', expr1, expr2) }
108         end
109
110         rule :less_than_or_equal_to do
111             match(:expr, '<', '=', :expr) { |expr1, _, _, expr2| Condition.new('<=', expr1,
    expr2) }
112             match(:expr, 'less', 'than', 'or', 'equal', 'to', :expr) { |expr1, _, _, _, _, _,
    expr2| Condition.new('less than or equal to', expr1, expr2) }
113         end
114
115         rule :greater_than_or_equal_to do
116             match(:expr, '>', '=', :expr) { |expr1, _, _, expr2| Condition.new('>=', expr1,
    expr2) }
```

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

```
    "endif") { |_, _, bool_log, _, _, if_states, _, else_states, _|
                If.new(bool_log, if_states, else_states) }
            end

        rule :print do
            match("write", :multiple_strings) { |_, mult_str| Print.new(mult_str) }
            match("write", :string_expr) { |_, str_exp| Print.new(str_exp) }
            match("write", :bool_logic) { |_, bool_log| Print.new(bool_log) }
            match("write", :expr) { |_, exp| Print.new(exp) }
            end

        rule :atom do
            match(:funcCall)
            match(Float) { |float_num| Constant.new(float_num) }
            match(Integer) { |int_num| Constant.new(int_num) }
            match("-", Float) { |a, b| Constant.new(b, a) }
            match("-", Integer) { |a, b| Constant.new(b, a) }
            match('(', :expr, ')') { |_,exp,_| Expression.new(exp) }
            match(:id)
            end

        end #end för alla rules
    #+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- END BNF +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    end #end för initialize

    def done(str)
        ["quit", "exit", "bye", "close", "stop"].include?(str.chomp)
    end

    #För att starta programmet i terminalen
    def activate_terminal
        print "[ETL] "
        str = gets
        if done(str) then
            puts "Bye."
        else
            parsePrinter = @etlParser.parse str
            puts "=> #{parsePrinter.eval}"
            activate_terminal
        end
    end

    #För att testa från en fil
    def activate_file(etl_file)
        @output = []
        etl_file = File.read(etl_file)
        @output = @etlParser.parse(etl_file)
        ##puts "=> #{output.eval}"
        @output
    end

    def log(state = true)
        if state
          @etlParser.logger.level = Logger::DEBUG
        else
          @etlParser.logger.level = Logger::WARN
        end
    end

end #end för klassen

checkEtl = Etl.new
checkEtl.log(false)
#checkEtl.activate_terminal
checkEtl.activate_file("etl.etl")
```

```
checkEtl.output.each { |segment|
  if segment.class != Function and segment.class != FunctionCall
     segment.eval()
    end }
```

6.3 classes.rb

```
## Alla klasser som behövs

$our_funcs = Hash.new

class ScopeHandler
    def initialize()
        @@level = 1
        @@holder = {}
    end
    def defineScope(s)
        @@holder = s
        return @@holder
    end
    def receiveHolder()
        return @@holder
    end
    def receiveLevel()
        return @@level
    end
    def incre()
        @@level = @@level + 1
        return @@holder
    end
    def decre(s)
        defineScope(s)
        @@level = @@level - 1
        return nil
    end
end

$scope = ScopeHandler.new


def look_up(variable, our_vars)
    levelNr = $scope.receiveLevel
    if our_vars == $scope.receiveHolder
        loop do
            if our_vars[levelNr] != nil and our_vars[levelNr][variable] != nil
                return our_vars[levelNr][variable]
            end
            levelNr = levelNr - 1
        break if (levelNr < 0)
        end

        if our_vars[levelNr] == nil
            our_vars[variable]
        end
    end
end

class Variable
    attr_accessor :variable_name
    def initialize(id)
        @variable_name = id
    end
    def eval
        return look_up(@variable_name, $scope.receiveHolder)
    end
end

class Expr
    attr_accessor :sign, :lhs, :rhs
    def initialize(sign, lhs, rhs)
```

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

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

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

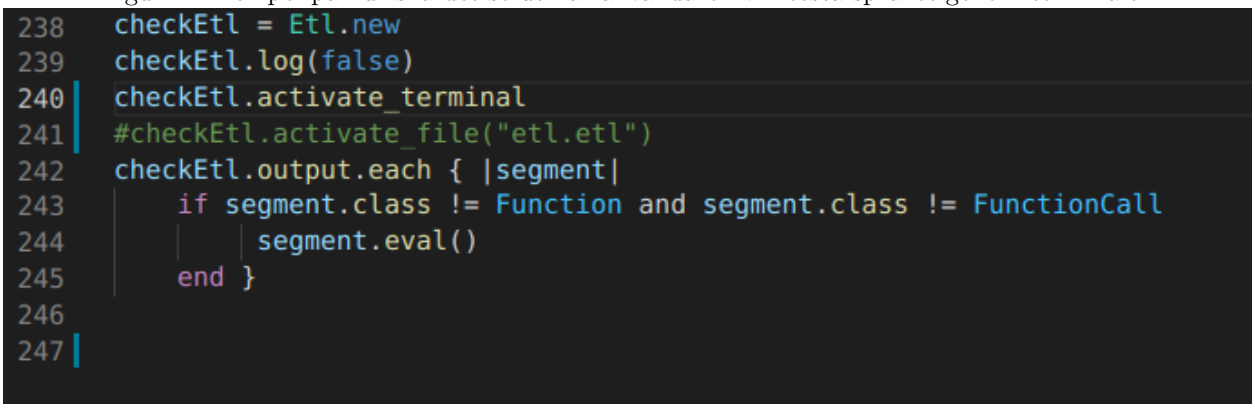
```
259             $our_funcs[def_name] = self
260         else
261             raise("OOOPS! THE FUNCTION \"#{@def_name}\" DOES ALREADY EXIST!")
262         end
263     end
264     def recieveStates()
265         @states
266     end
267     def recieveArgs()
268         @f_arguments
269     end
270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

6.4 etl.etl

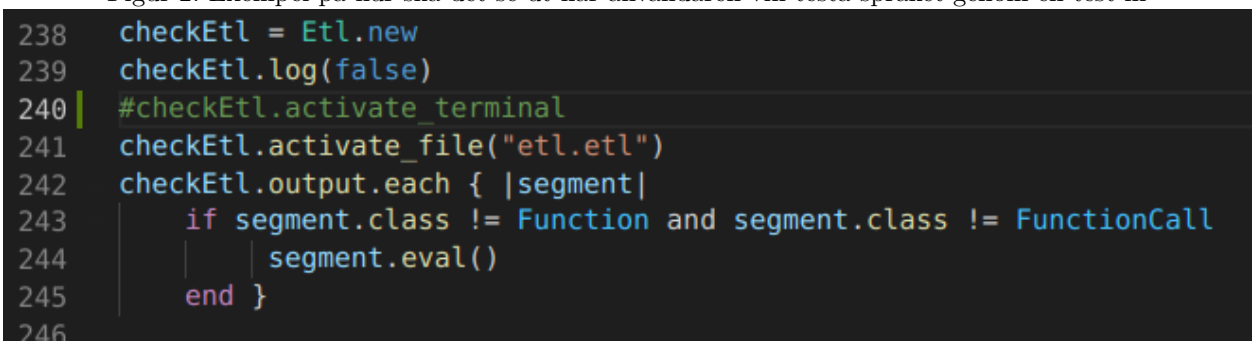
```
y = 1
while ( y < 4)
write "while loop fungerar"
y = y + 1
<<stop
endwhile


x = 7
u = 8
if (x > 6 and u equal 8) then
write "if sats fungerar"
otherwise
write "otherwise fungerar"
endif


c = 10

define add()
a = 4
b = 5
w = 99999999999
c = a + b
return c
enddef
write add()

write c
write w

s = 2

define foo(w, k)
s = w + k
j = 456456456456
return s
enddef
write foo(25, 75)

write s
write j


n = "Ahmed Sikh"
b = "Ismail"
v = " !!"

write "Hej" plus " på dig"

write "ETL" plus " är" plus " lätt"

write b plus v

write n plus v plus b
```

7 Bilder

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



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



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