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

4 maj 2021

Innehåll

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första version av Språkdokumentation	210430

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

kanske med scopehantering (vi får se)

6 BNF Grammatik

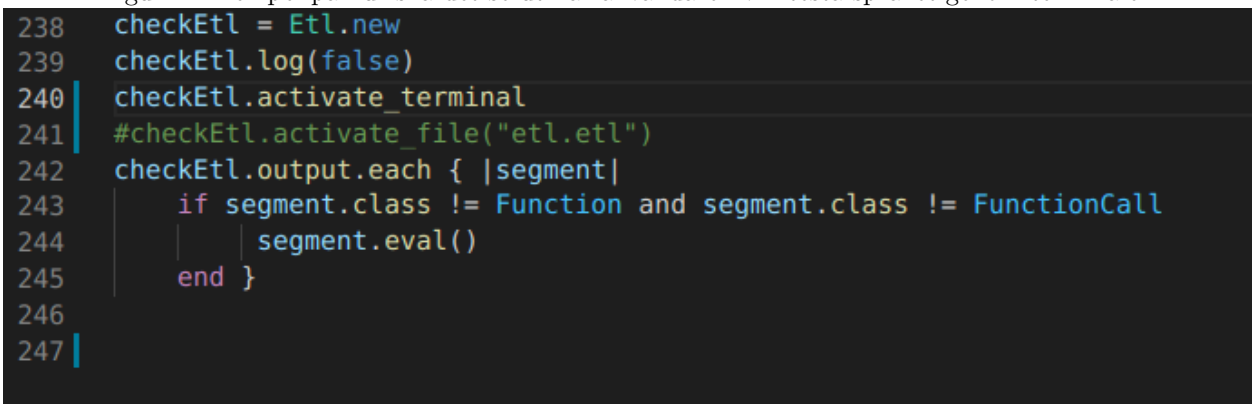
```
<PROGRAM>       ::= <STATEMENTS>

<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>
                | <STRING_EXPR> plus <STRING_EXPR>
##########################################################

<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>

<NOT_EQUAL_TO>::= <EXPR> != <EXPR>
                  | <EXPR> not equal to <EXPR>

<EQUAL>         ::= <EXPR> == <EXPR>
                  | <EXPR> equal <EXPR>

<ID>            ::= /[a-z]+[a-z0-9_]*/

<FUNC>          ::= define /[a-z]+[a-z0-9_]*/ ( <ARGUMENTS> ) <STATEMENTS> enddef
```

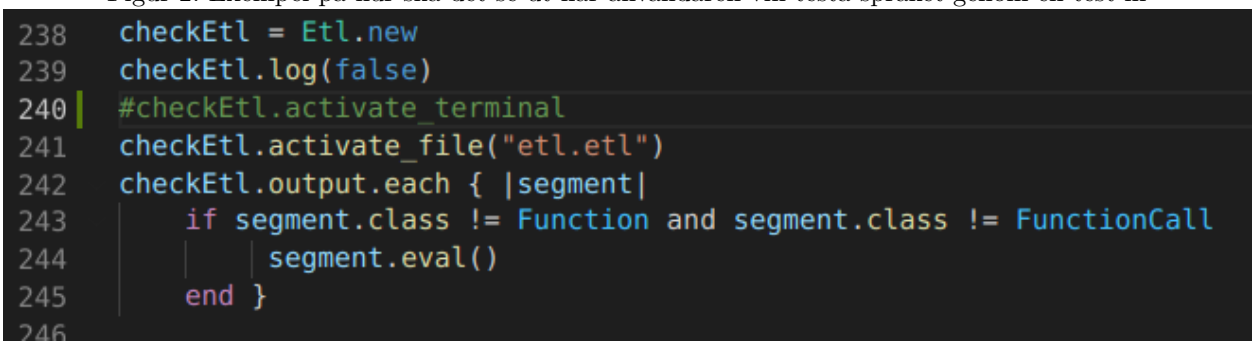
```
                | define /[a-z]+[a-z0-9_]*/ ( ) <STATEMENTS> enddef

<FUNCCALL>      ::= <ID> ( )
                | <ID> ( <ARGUMENT> )

<RETURN>        ::= return <ARGUMENT>

<ARGUMENTS>     ::= <ARGUMENTS> , <ARGUMENT>
                | <ARGUMENT>

<ARGUMENT>      ::= <MULTIPLE_STRINGS>
                | <STRING_EXPR>
                | <EXPR>
                | <BOOL_LOGIC>

<WHILE_LOOP>    ::= while ( <BOOL_LOGIC> ) <STATEMENTS> endwhile

<STOP>          ::= stop

<IF_BOX>        ::= if ( <BOOL_LOGIC> ) then <STATEMENTS> endif
                | if ( <BOOL_LOGIC> ) then <STATEMENTS> otherwise <STATEMENTS> endif

<PRINT>         ::= write <MULTIPLE_STRINGS>
                | write <STRING_EXPR>
                | write <BOOL_LOGIC>
                | write <EXPR>

<ATOM>          ::= <FUNCTION_CALL>
                | <Float>
                | <Integer>
                | - <Float>
                | - <Integer>
                | ( <EXPR> )
                | <ID>
```

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**.