



ETL

Easy To Learn

TDP019 Projekt: Datorspråk

Språkdokumentation

Författare

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

Innehåll

1	Revisionshistorik	2
2	Inledning	2
2.1	Syfte	2
2.2	Introduktion	2
2.3	Målgrupp	2
3	Användarhandledning	2
3.1	Installation	2
3.2	Variabler och Tilldelning	3
3.3	Matematiska Operationer	3
3.4	Kommentarer	3
3.5	Print	4
3.6	Villkor/If-satser	4
3.7	Iteration	5
3.8	Funktioner	6
3.9	Multiple Strings	7
4	Systemdokumentation	8
4.1	Lexikaliska Analys	8
4.2	Parsning	9
4.3	Kodstandard	9
5	Reflektion	10
6	Bilagor	11
6.1	BNF Grammatik	11
6.2	ETL.rb	13
6.3	classes.rb	17
6.4	etl.etl	22
7	Bilder	23

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första version av Språkdokumentation	210507

2 Inledning

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

2.1 Syfte

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

2.2 Introduktion

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

2.3 Målgrupp

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

3 Användarhandledning

3.1 Installation

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

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

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

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

Det finns två sätt att köra ETL språket på:

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

3.2 Variabler och Tilldelning

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

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

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

3.3 Matematiska Operationer

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

Exempel:

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

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

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

3.4 Kommentarer

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

Exempel på flerradskommentar:

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

Exempel på enkelradskommentar:

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

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

3.5 Print

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

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

Skriver ut följande:

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

3.6 Villkor/If-satser

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

Exempel på en if-sats:

```
x = 7
y = 8
if (x > 6 and y == 8) then
write "if-sats 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 tom parentes så att kodraden kommer se ut så här: **define name()** i slutändan.

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

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

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

Exempel på funktioner utan parameter:

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

write add()
```

Skriver ut följande:

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

2. Funktioner med parametrar:

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

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

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

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

Exempel på funktioner med parameter:

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

write add(25, 75)
```

Skriver ut följande:

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

3.9 Multiple Strings

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

Exempel 1:

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

Skriver ut följande:

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

Exempel 2:

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

Skriver ut följande:

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


4 Systemdokumentation

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

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

4.1 Lexikaliska Analys

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

I slutet kommer alla tokens skickas vidare till parsen.

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

1. Tokens som inte ska parsas och kommer ignoreras:

- Matchar och ignorerar flerradskommentarer.

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

- Matchar och ignorerar enkelradskommentar

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

- Matchar och ignorerar alla mellanrum

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

2. Tokens som ska parsas:

- Matchar alla flyttal och returneras som Float”

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

- Matchar alla heltal och returneras som "Integer”

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

- Matchar strängar inom enkelcitattecken

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

- Matchar strängar inom dubbelcitattecken

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

- Matchar namn på variabler

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

- Matchar allt annat(enkla kårakterer)

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

4.2 Parsning

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

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

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

4.3 Kodstandard

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

Vissa kodstandard som **ETL** har:

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

5 Reflektion

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

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

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

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

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

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

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

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

6 Bilagor

6.1 BNF Grammatik

```

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

```

```

62 <GREATER_THAN_OR_EQUAL_TO> ::= <EXPR> >= <EXPR>
63     | <EXPR> greater than or equal to <EXPR>
64
65 <NOT_EQUAL_TO> ::= <EXPR> != <EXPR>
66     | <EXPR> not equal to <EXPR>
67
68 <EQUAL>         ::= <EXPR> == <EXPR>
69     | <EXPR> equal <EXPR>
70
71 <ID>            ::= /[a-z]+[a-z0-9_]*/
72
73 <FUNC>          ::= define /[a-z]+[a-z0-9_]*/ ( <ARGUMENTS> ) <STATEMENTS> enddef
74     | define /[a-z]+[a-z0-9_]*/ ( ) <STATEMENTS> enddef
75
76 <FUNCCALL>      ::= <ID> ( )
77     | <ID> ( <ARGUMENT> )
78
79 <RETURN>        ::= return <ARGUMENT>
80
81 <ARGUMENTS>     ::= <ARGUMENTS> , <ARGUMENT>
82     | <ARGUMENT>
83
84 <ARGUMENT>      ::= <STRING_EXPR>
85     | <EXPR>
86
87 <WHILE_LOOP>    ::= while ( <BOOL_LOGIC> ) <STATEMENTS> endwhile
88
89 <STOP>          ::= stop
90
91 <IF_BOX>        ::= if ( <BOOL_LOGIC> ) then <STATEMENTS> endif
92     | if ( <BOOL_LOGIC> ) then <STATEMENTS> otherwise <STATEMENTS> endif
93
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>

```



```

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

```

```

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

```


16 / 24

6.3 classes.rb

```
1 ## Alla klasser som behövs
2
3 $our_funcs = Hash.new
4
5 class ScopeHandler
6   def initialize()
7     @@level = 1
8     @@holder = {}
9   end
10  def defineScope(s)
11    @@holder = s
12    return @@holder
13  end
14  def receiveHolder()
15    return @@holder
16  end
17  def receiveLevel()
18    return @@level
19  end
20  def incre()
21    @@level = @@level + 1
22    return @@holder
23  end
24  def decre(s)
25    defineScope(s)
26    @@level = @@level - 1
27    return nil
28  end
29 end
30
31 $scope = ScopeHandler.new
32
33
34 def look_up(variable, our_vars)
35   levelNr = $scope.receiveLevel
36   if our_vars == $scope.receiveHolder
37     loop do
38       if our_vars[levelNr] != nil and our_vars[levelNr][variable] != nil
39         return our_vars[levelNr][variable]
40       end
41       levelNr = levelNr - 1
42       break if (levelNr < 0)
43     end
44
45     if our_vars[levelNr] == nil
46       our_vars[variable]
47     end
48   end
49 end
50
51 class Variable
52   attr_accessor :variable_name
53   def initialize(id)
54     @variable_name = id
55   end
56   def eval
57     return look_up(@variable_name, $scope.receiveHolder)
58   end
59 end
60
61 class Expr
62   attr_accessor :sign, :lhs, :rhs
63   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     when '^'
79       return lhs.eval ** rhs.eval
80     when '%'
81       return lhs.eval % rhs.eval
82     else nil
83   end
84 end
85 end
86
87 class Plus_str
88   attr_accessor :sign, :lhs, :rhs
89   def initialize(sign, lhs, rhs)
90     @sign = sign
91     @lhs = lhs
92     @rhs = rhs
93   end
94   def eval()
95     case @sign
96     when 'plus'
97       return @lhs.eval + @rhs.eval
98     else nil
99   end
100 end
101 end
102
103 class Condition
104   attr_accessor :sign, :lhs, :rhs
105   def initialize(sign, lhs, rhs)
106     @sign = sign
107     @lhs = lhs
108     @rhs = rhs
109   end
110   def eval()
111     case sign
112     when '<', 'less than'
113       return lhs.eval < rhs.eval
114     when '>', 'greater than'
115       return lhs.eval > rhs.eval
116     when '<=', 'less than or equal to'
117       return lhs.eval <= rhs.eval
118     when '>=', 'greater than or equal to'
119       return lhs.eval >= rhs.eval
120     when '!=', 'not equal to'
121       return lhs.eval != rhs.eval
122     when '==', 'equal'
123       return lhs.eval == rhs.eval
124     when 'and'
125       return lhs.eval && rhs.eval
126     when 'or'
127       return lhs.eval || rhs.eval
128     else nil

```

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

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

```

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

```

6.4 etl.etl

```
1 y = 1
2 while (y < 5)
3 write "while loop fungerar"
4 y = y + 1
5 <<stop
6 endwhile
7
8
9 x = 7
10 u = 8
11 if (x > 6 and u equal 8) then
12 write "if-sats fungerar"
13 otherwise
14 write "otherwise fungerar"
15 endif
16
17
18 c = 10
19
20 define add()
21 a = 4
22 b = 5
23 w = 9999999999
24 c = a + b
25 return c
26 enddef
27 write add()
28
29 write c
30 write w
31
32 s = 2
33
34 define foo(w, k)
35 s = w + k
36 j = 456456456456
37 return s
38 enddef
39 write foo(25, 75)
40
41 write s
42 write j
43
44
45 n = "Ahmed Sikh"
46 b = " Ismail"
47 v = " !"
48
49 write "Hej" plus " på dig"
50
51 write "ETL" plus " är" plus " lätt"
52
53 write b plus v
54
55 write n plus v plus b
56
57
58 write 5 ^ 2
59
60 write 10 % 3
```

7 Bilder

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

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

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

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


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

