LINKÖPINGS UNIVERSITET

TDP019

Ezprog Dokumentation

Författare

William Sjöström, wilsj389@student.liu.se
Sebastian Grunditz, sebgr273@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018
Version 1.1

2019-05-14

Innehåll

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Skapade Dokumentation för språket Ezprog	190514

2 Inledning

Ezprog är ett ganska simpelt programmeringsspråk som har tagit de bästa delarna (enligt oss) från flera andra språk som c++, Python, och ruby. Detta språk gjordes under andra terminen på IP-programmet vid Linköpings universitet tillhör kursen TDP019 Projekt: Datorspråk.

2.1 Syfte

Syftet med denna kursen och projektet vi utförde i den var att vi skulle fördjupa oss i hur språk fungerar samt är uppbyggda på en lite djupare nivå.

2.2 Introduktion

Som nämnt lite i inledningen så var vår plan med detta språk var att kombinera dom enligt oss bästa funktionella delarna från flera olika språk. Samtidigt vill vi att språket skall vara relativt nybörjarvänligt och vi har därför gjort mycket syntax som är väldigt likt vanligt skriftspråk.

2.3 Målgrupp

Målgruppen till vårt språk är nybörjare som och vi har då egentligen ingen specifik demografi som språket är riktat åt.

3 Användarhandledning

3.1 Installation

För att kunna köra vårt språk behöver man utföra några enkla steg. För de första måste ruby vara installerat. Sedan får man ändra i filen run.rb och skriva in det filnamnet till filen man vill köra mellan parenteserna. ”./” måste skrivas innan filnamnet. Exempel “./test.txt”. Sedan skriver du “ruby run.rb” i terminalen för att köra filen.

3.2 Variabler och Tilldelning

Variabler i Ezprog är dynamiskt tilldelade behållare för data, gjorda för att användaren inte ska behöva specificera vad det är för datatyp som ska lagras. Den enda regel som finns för variabler är restriktioner på namn, vilket gås igenom nedan. Exempel på variabeltilldelning följer nedan:

```
name = 'Sebastian'
age = 22
height = 183.0
weight = calcWeight(height)
```

Som syns behövs det inte anges vilken datatyp som sparas, utan bara ett namn och ett värde. Formen på variabelskapande är:

```
<Variabelnamn> = <expression>
<Variabelnamn> = <function_call>
```

Det går alltså även att tilldela resultat av funktionsanrop till variabler, om den funktion som anropas har en retur-sats. Reglerna för variabelnamn i Ezprog är enkla och korta; variabelnamn får endast innehålla gemener a-z, för att tvinga användarna av språket att komma på variabelnamn som inte innehåller en massa onödiga tecken.

Tilldelning med matematiska operatorer fungerar på samma sätt, fast då med vissa restriktioner på vilka typer det är som ska räknas ut. Exempel: sträng += annansträng tal += 1 flyttal *= annatflyttal tal /= flyttal

Den största restriktionen för tilldelning med matematiska operatorer är att de inte fungerar för booleska uttryck överhuvudtaget, och bara += fungerar för sträng; heltal och flyttal kan dock använda sig av alla matematiska tilldelningsoperatorer.

Tilldelning i Ezprog med matematiska operatorer fungerar inte i nuläget.

3.3 Matematiska Operationer

Matematiska operationer utförs som följande i “”. Syntaxen till matematiska operationer är väldigt lik skriftlig matematik. Exempel skulle kunna vara:

```
1 + 1
1 - 1
1 * 2
2 / 1
```

Man kan också använda matematiska operationer för att utföra ändra på variabler av formen Integer eller Float. Exempel:

```
x = 1
x = x - 1
```

Detta skulle då leda till att x = 0. Samma skulle fungera för de olika exempel som visas ovan. För lite mer komplexa matematiska operationer så följer språket de mest grundläggande matematiska reglerna som att gånger utförs före addition till exempel. Så x = 6 - 4 * 2 - 1 skulle då leda till att x = -3.

3.4 Kommentarer

I Ezprog finns det både flerradskommentarer, och enkelradskommentarer. Ingen av kommentar-sorterna är svåra att skriva, men de skrivs på lite olika sätt. För att skriva en multi-line kommentar börjar man med !comment, sedan skriver man det som ska stå i kommentaren och avslutar med !end. Exempel:

```
!comment
    Det här är en flerradskommentar. '12378182*12235'?
    det går att skriva i princip vad som helst här inne, så länge man inte skriver utropste
    end innan man vill avsluta den.
!end
```

Det enda tecken man inte får skriva inne i en flerradskommentar är utropstecken, då det kommer att säga till programmet att det kommer ett end efter, och att det är dags att avsluta kommentaren.

För att skriva en enkelradskommentar skriver man två utropstecken, följt av det man vill kommentera. Exempel:

```
!!Det här är en enkelradskommentar.
!!Den hanterar inte flera rader på en gång.
!!Så för varje rad får man börja med !! på nytt.
```

Enkelradskommentarer är simplare då de inte innehåller restriktioner på vilka tecken som får vara inne i kommentaren, utan bara att de ska börja med dubbla utropstecken. Kommentarer är främst till för att låta utvecklare dokumentera vad funktioner och kodblock gör, flytande i koden samt lämna kommentarer och TODO för framtida skådande.

3.5 Print

Utskrift i Ezprog är väldigt simpelt, då det bara är att skriva print(, följt av det man vill skriva ut, och avsluta med). I Ezprog kan man skriva ut alla sorts datatyper, men även variabler. Exempel:

```
Print(5) =>5

Print('Hej hej') => Hej hej

tre = 3
Print(tre) => 3
```

Utskrift skriver alltid ut det aktuella värdet på en variabel, så om variabeln tre skulle ändra värde till 5 innan den skrivs ut, kommer Print att skriva ut 5 istället för 3.

Man kan i nuläget inte skriva ut resultat av funktionsanrop direkt, utan det måste först sparas till en variabel innan det skrivs ut. Det går inte heller att skriva ut resultat av booleska jämförelser i nuläget.

3.6 Vilkorssatser

För att göra villkorssatser är uppbyggda som följande. Man skriver If följt av en öppnande parentes, sedan skrivs den eller de olika logiska uttrycken som villkorssatsen skall evaluera. Sedan skriver man en avslutande parentes följt av en öppnande måsvinge (). Sedan skrivs de olika uttryck som villkorssatsen skall utföra om villkoret är sant. Efter alla uttryck skrivs en avslutande måsvinge.

Exempel på en villkorssats är:

```
If(3 is less than 4 and 1 is equal to 1){
    Print“(”Hej)
}
```

För att göra else is satser Gör man precis som för villkorssatsen men istället skriver Elseif istället för If. Else if måste skrivas precis efter en villkorssats för att fungera. Else måste skrivas efter If eller Elseif om det finns en Elseif. Else skrivs bara som Else utan några Exempel är:

```
If(3 is less than 4 and 1 is equal to 1){
    Print('Hej1')
}
Elseif(3 is greater than 2 and 2 is greater or equal to 1){
    Print('Hej2')
}
Else{
    Print('Hej3')
}
```

3.7 Loopar

Ezprog är uppbyggt av 3 olika sorters loopar. Dessa loopar är while, for, och times. En while loop är lite uppbyggt som en if sats. För att göra en While loop skriver man först While och sedan en öppnande parentes för villkoren. Sen skriver man villkoren följt av en stängande parentes. Till sist skriver man alla argument inuti måsvingar. Exempel på while loop:

```
x = 10
Loop while x is greater or equal to 0{
    Print(x)
    x = x - 1
}
```

For loopar skrivs som följande. Man börjar på samma sätt som på alla loopar med att skriva Loop och sedan ett variabelnamn. Sedan skriver man from följt av ett startvärde. Sedan skriver man to följt av ett slutvärde. Exempel på en for loop ser ut som följande:

```
Loop for x from 0 to 10{
    Print(x)
}
```

Till sist har vi times loopar som loopar ett x antal gånger. Man skriver då bara Loop följt av antal gånger man vill loopa och sedan times. Exempel på times loop:

```
Loop 10 times{
    Print('hej')
}
```

3.8 Funktioner

Funktioner i Ezprog är väldigt simpla att skapa, det svåraste kan vara att komma på ett passande funktionsnamn. Funktioner skapas på formeln:

```
define <funktionsnamn>(funktionsvariabler){<statements>}
```

Det liknar variabeltilldelning i att man inte behöver specificera vilken typ av data det är som kommer returneras, för att göra det lite enklare att ändra på vad som returneras. Användaren behöver inte heller specificera vilken datatyp funktionsvariablerna ska ha, utan bara ge dem ett namn som kommer användas inne i funktionskroppen. Exempel:

```
define PrintNumTimes(number, x){
    loop x times {
    Print(number)
    }
}
```

Funktionen tar in två variabler: number och x. Sedan använder den x för att loopa över, så att den skriver ut number så många gånger som x är stort. I den här funktionen finns det ingen retursats, vilket innebär att det inte går att tilldela ett anrop på denna funktion till en variabel.

```
define Fibonacci(stop){
   If (stop is equal to  0){
      return 1
   }
   Else{
       a = 1
       b = 1
       loop stop-1 times{
                a = b
        b = a+b
       }
    return a
}
```

Det här är en lite mer avancerad version av en funktion. Det är en iterativ implementation av en funktion för att beräkna Fibonaccital. Den tar in en variabel, start, som sedan används för att kolla hur många gånger vi ska loopa, för att få fram fibonaccitalet på positionen F[start]. Här finns även en retursats i slutet, vilket gör att resultatet kan skrivas ut med en printsats eller tilldelas till en variabel.

4 Systemdokumentation

4.1 Lexikalisk analys

Eftersom vi använder oss av rdparse-parsern för vårt språk har vi byggt den lexikaliska analysen på tokenisering av reguljära uttryck i programmeringsspråket Ruby. Med hjälp av dessa regler kan vi sålla ut relevanta token i den programkod som skrivits, för att överlåta till Ezprog för tolkning. De tokens som finns är följande, i ordning:

Matchar flerradskommentarer och ignorerar dessa, då kommentarer inte ska parsas i vårt språk.

```
token(/\!comment(.+?|\n)\!end/m) #ignores multiline comments
```

Denna matchning gör enradskommentarer till tokens, och ignorerar dessa, precis som flerradskommentarer.

```
token(/(!!.+$)/) #ignores single line comments
```

Matchar olika slags blanksteg och ignorerar dessa; detta är inte ett esoteriskt språk som bryr sig om dessa, utan endast de nyckelord vi valt ut.

```
token(/\s+/) #ignores whitespaces
```

Här matchas datatyper och returneras korrekt.

```
token(/\d+/) {|m| m.to_i } #match integer
token(/"[^\"]*"/) {|m| m } #match "string"
token(/'[^\']*'/) {|m| m } #match 'string'
```

Matchar olika nyckelord, som kan användas för att bygga upp ett parsningsträd.

```
token(/Print/) {|m| m } #match print_stmts
token(/Loop/) {|loop| loop} #match loop_stmt
token(/is/) #part of compare_handler
token(/If\(.+?\){.+?}/m) {|iff| iff } #match if_stmt
token(/Elseif/) {|elseif| elseif } #match elseif_stmt
token(/Else/) {|els| els } #match else_stmt
token(/Return/) {|retur| retur} #match return_stmt
```

Matchar diverse specialtecken som kanske inte faller under . i Rubys regex.

```
token(/(\+=|-=|\*=|\/=|\+\+|--|\+|\-|\*|\/|!=|\.|,|\]|\[|{|}|\(|\)|\:|<=|>=|<|>|==|=)/)
{|m| m}
```

Matchar variabelnamn, så att vi kan använda dem för att göra variabler.

```
token(/([a-z]+)/) {|m| m } #match variable name
```

Matchar de resterande tecken som inte tagits upp innan, så som smådelar av andra satser.

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

4.2 Parsning

Parsningen börjar efter att texten har delats upp i tokens. Med dessa tokens delas koden upp och gör klassvariabler av dem för att sedan evalueras. Parsern funkar rekursivt och och går neråt enligt BNFen.

5 Reflektion

Det första momentet i kursen, att välja vad vi ville göra för programmeringsspråk, visade sig vara den svåraste utmaningen under kursens gång. Det tog ett flertal veckor innan vi kom fram till en tillräckligt annorlunda, utmanande och spännande idé för att sätta det i verket. När väl det bestämts var det relativt lätt att sätta sig ner och bestämma hur vi tyckte att språket skulle se ut, främst med språkexempel som vi i slutändan vill att språket ska kunna klara av genom att köra utan ändring.

Med språkexempel klara och en vilja att börja utveckla var det dags att börja utveckla BNFen till språket. Det visade sig också svårt, då det är krångligt att tänka ut en väg som språket ska traversera i det lexikaliska trädet, utan att sitta ner och programmera samtidigt. Om man jämför den arma version av BNF som finns i den språkspecifikation vi skrivit med den som vi skrivit i BNF, kan man se att det utvecklats rätt mycket; det är även stor skillnad på dessa, med endast ströbitar som stämmer överens mellan båda.

Det som stämmer mest överens med den vision vi hade i början av projektet är hur språket ser ut. Skillnaderna mellan den teoretiskt skrivna kod som står i språkspecifikationen och det som går att skriva i Ezprog är väldigt små; den största skillnaden är att vi inte implementerat listor.

När man börjar med ett projekt är det lätt att vara ambitiös med vad det ska kunna göra; man ser alltid en stor vision om hur det ska bli det nästa stora. Dock inser man snabbt att det inte är lätt att skapa stora projekt, om man inte lägger mycket tid på det. Som ovan nämnt var det största vi inte hann med listor, då det inte fanns tid för det. Hade vi haft en vecka till på oss hade vi kunnat implementera det, men med lite felprioritering av tid så innebar det att vi inte hade nog med tid.

En av de enklare delarna med att göra ett eget språk var, konstigt nog, de svårare delarna: loopar och funktioner. Vi hade problem, men jämfört med hur stor del av ett programmeringsspråk de delarna är, var det relativt relativt lätt att implementera. Den enda delen av funktioner vi inte lyckades implementera var rekursion.

Trots alla motgångar så har det varit väldigt intressant att få implementera ett eget programmeringsspråk. Kanske kommer vi aldrig att jobba med det igen, men om någon frågar oss om vi kan, kan vi iallafall säga att vi försökte en gång.

6 BNF

```
<run>
        ::= <stmt_list>

<stmt_list>
    ::= <stmt>
        |<stmt_list> <stmt>

<stmt>
    ::= <if_stmt>
        |<print_stmt>
        |<func_stmt>
        |<loop_stmt>
        |<return_stmt>
        |<assign_stmt>

<print_stmt>
    ::= ""Print ""( <expr> "")

<return_stmt>
    ::= ""Return <func_call>
         | ""Return <expr>

<assign_stmt>
    ::= /[a-z]+/ ""= <func_call>
        |/[a-z]+/ ""= <expr>

<if_stmt>
    ::= <if_rule>

<if_rule>
    ::= <if> <elsif> <els>
        |<if> <elsif>
        |<if> <els>
        |<if>

<if>
    ::= ""If ""( <bool_logic> "") ""{ <stmt_list> ""}

<elsif>
    ::= ""Elseif ""( <bool_logic> "") ""{ <stmt_list> ""}
          |<elsif> <elsif>

<els>
    ::= ""Else ""{ <stmt_list> ""}
```

```
<loop_stmt>
    ::= <loop_rule>

<loop_rule>
    ::= <while_loop>
        |<for_loop>
        |<times_loop>

<while_loop>
    ::= Loop while <bool_logic> { <stmt_list> }

<for_loop>
    ::= Loop <loop_variable> from <expr> to <expr> { <stmt_list> }

<times_loop>
    ::= Loop <expr> times { <stmt_list> }

<loop_variable>
    ::= /[a-z]+/

<bool_logic>
    ::= <logic_operand> <logic_op> <logic_operand>
        |<bool_logic> <logic_op> <logic_operand>
        |<logic_operand>
        |<bool>

<logic_op>
    ::= and
          |or

<logic_operand>
    ::= <bool_expr>
        |<bool>

<bool_expr>
    ::= <expr> greater than or equal to <expr>
        |<expr> less than or equal to <expr>
        |<expr> less than <expr>
        |<expr> greater than <expr>
        |<expr> equal to <expr>
        |<expr> not equal to <expr>

<func_stmt>
    ::= <func_init>
        |<func_call>

<func_init>
    ::= define /[a-z+]/ ( <args> ) { <stmt_list> }

<func_call>
    ::= <func_var> ( <params> )
```

```
<func_var>
    ::= /[a-z]+/

<args>
    ::= <args> , <arg>
        |<arg>

<arg>
    ::= <expr>

<params>
    ::= <params> , <param>
        |<param>

<param>
    ::= <func_call>
        |<expr>

<expr>
    ::= <expr> + <term>
        |<expr> - <term>
        |<term>

<term>
    ::= <term> * <atom>
    |<term> / <atom>
    |<atom>

<atom>
    ::= <variable>
        |<datatype>
        |( <expr> )

<variable>
    ::= /[a-z]+/

<datatype>
    ::= <bool>
        |Integer
        |Float
        |String

<bool>
    ::= True
        |False
```