1. Inledning

Det här är dokumentationen till datorspråket *Pseudocode* som har skapats i kursen TDP019 av David Svenson och Olle Kvarnström, studenter på programmet Innovativ programmering vid Linköpings universitet. Språket har som mål att fungera som ett pseudospråk men att dessutom faktiskt kunna köras. Det innehåller nästan inga symboler, utan de flesta är ersatta av ord. Att programmera i *Pseudocode* blir därför lite som att skriva flytande engelsk text. Språket riktar sig till personer som snabbt vill kunna skriva prototyper och dessutom få ut något resultat av det. Det är då lättare att visa upp koden för en person som inte är insatt i programmering. *Pseudocode* kan också vara ett alternativ för personer som är nybörjare på programmering eftersom syntaxen är mer naturlig och lik vardagsspråk än andra datorspråk.

2. Användarhandledning

2.1 Allmänt

Pseudocode körs antingen genom att anropa språkets interpretator och låta den läsa från en källkodsfil, eller interaktivt genom att skriva till interpretatorn rad för rad i en terminal. Det interaktiva läget stödjer för tillfället inte uttryck som sträcker sig över flera rader. Språket är imperativt med stöd för funktioner, det använder sig av dynamisk typning och indentering är en viktig del av syntaxen.

2.1.1 Syntax

Programkod som skrivs i *Pseudocode* består av noll eller flera satser som separeras med nyradstecken. Det finns satser som sträcker sig över flera rader där den första raden (satsens huvud) avgör om de efterföljande ska utföras eller inte. Då gäller det att de tillhörande raderna indenteras mer än satshuvudet. För att se exempel på detta kan du kolla på *2.3.7. Villkorssatser* nedan.

Kommentering i *Pseudocode* görs per rad med brädgårds-tecknet (#). Allt som finns efter ett sånt tecken på en rad ignoreras när koden körs.

2.1.2 Om handledningen

I användarhandledningens kodexempel används “>>” för att markera inmatning till interpretatorn. Resultatet av det inmatade uttrycket visas på nästa rad.

2.2 Språkets datatyper

2.2.1 Tal

I *Pseudocode* omfattar taltypen både det som inom programmering kallas flyttal/decimaltal (*floating point*) och heltal (*integer*). Alla tal kan vara både positiva och negativa. Räknereglerna är desamma för alla tal men det finns vissa skillnader i hur de får användas i språkets olika konstruktioner.

2.2.2 Strängar

Strängar skrivs mellan citationstecken och använder sig av teckenkodningen UTF-8.

Exempel:

```
“Hello world!”
```

2.2.3 Listor

En lista i *Pseudocode* kan bestå av godtyckligt många element av godtycklig typ. Den avgränsas med hakparenteser och elementen separeras av kommatecken. Listor kan även innehålla uttryck.

Exempel:

```
[1, 2, 3, ”hej”, 5, [1, 2, 4]]
[1, 2 , 1 plus 2]
```

2.3 Språkkonstruktioner

2.3.1 Aritmetiska uttryck

Till skillnad från de flesta andra programmeringsspråk så använder *Pseudocode* ord för de aritmetiska operatorerna istället för de klassiska symbolerna; +, -, etc. Operatorerna fungerar på snarlika sätt för de olika datatyperna. De operationer som finns i *Pseudocode* är addition, subtraktion, multiplikation, division och modulo. De illustreras i exemplen här nedanför.

Exempel:

```
>> 1 plus 2
3
>> 4 minus 7
-3
>> 2 times 6
12
>> 10.0 divided by 3
3.3333333333333335
>> 6 modulo 3
0

>> “ett” plus “två”
“etttvå”
>> “ett” minus 1
“et”
>> “ett” times 4
“ettettettett”

>> [1,2,3] plus 4
[1, 2, 3, 4]
>> [1,2,3] plus [4]
[1, 2, 3, [4]]
>> [1,2,3] minus 2
[1]
>> [1,2] times 2
[1, 2, 1, 2]
```

2.3.2 Boolska uttryck

Ett boolskt uttryck är antingen sant eller falskt. Värdena representeras av orden *true* och *false*. Det går också att använda tal i boolska uttryck och då tolkas talet 0 som falskt medan alla andra tal tolkas som sant.

Exempel:

```
>> true and false
false
>> true and true
true
>> true or false
true
>> false or false
false

>> 10.0 and true
true
>> true or 0
true

>> not false or false
true
```

2.3.3 Variabler

För att spara resultatet av ett uttryck eller något konstant värde tilldelar man det till en variabel. Variabler kan bara tilldelas ett värde av vilken datatyp som helst, men bara ett åt gången, då följande tilldelningar till samma variabel skriver över det föregående värdet. Variabelnamn måste bestå av stora och/eller små bokstäver från a till z.

Exempel:

```
>> myVar equals 4
4
>> myVar plus 52
56
```

Det går att förändra en variabels värde genom addition, subtraktion, multiplikation och division.

Exempel:

```
>> myVar equals 4
4
>> increase myVar by 2
6
>> decrease myVar by 1
5
>> multiply myVar by 2
10
>> divide myVar by 5
2
```

2.3.4 Utskrift till terminal

Resultat av alla uttryck kan skrivas ut till terminal. För att göra det används kommandot nedan:

Exempel:

```
>> write 4 plus 4
8
```

Kommandot “write” kan ta emot en lista av argument att skriva ut. Efter det sista argumentet skrivs ingen radbrytning ut. Det går dock att skicka strängen “\n” till kommandot för att manuellt göra en radbrytning.

Exempel:

```
>> write 4 plus 4
8>> write 4 plus 4, “\n”
8
>> myVar equals “Hello world!”
“Hello world!”
>> write myVar, “\n”
“Hello world!”
```

2.3.5 Inmatning av data

Det går att mata in data till en variabel genom kommandot “read to”. Variabeln behöver inte vara deklarerad sedan tidigare. Datan som matas in tolkas av *Pseudocode* som en sträng.

Exempel:

```
>> read to myVar
inmatad data              # inmatning
“inmatad data”            # resultat av inmatning
>> read to myVar
5156                      # inmatning
“5156”                    # resultat av inmatning
```

2.3.6 Jämförelser

Det går att jämföra aritmetiska uttryck, strängar och listor med varandra. Jämförelser används generellt i villkorssatser och loopar och ger alltid *true* eller *false* som resultat.

Exempel:

```
>> 4 is less than 5
true
>> 4 is greater than 5
false
>> 10 is 5 or more
true
>> 10 is 5 or less
false
>> 6 is between 5 and 7
true
>> 4 is 6
false
>> “ett” is “två”
false
```

2.3.7 Villkorssatser

Den traditionella typen av “if”-satser finns i *Pseudocode*. Den enklaste formen består bara av en “if”-gren. Efter en “if”-gren kan flera “else if” och till slut en “else” följa. Varje gren i en villkorssats ges ett uttryck och om det är sant utförs de tillhörande (indenterade) kodraderna.

Exempel:

```
#! /usr/bin/env pseudocode

if true then
  write “First if”
else if true then
  write “First else if”
else
  write “else”
```

```
Resultat: First if
```

```
#! /usr/bin/env pseudocode

if false then
  write “First if”
else if false then
  write “First else if”
else
  write “else”
```

```
Resultat: else
```

2.3.8 Upprepningssatser

Det finns två typer av upprepningssater:

while

...utför tillhörande kodrader så länge uttrycket i satshuvudet är sant.

Exempel:

```
#! /usr/bin/env pseudocode

number equals 5

while number is greater than 0 do
  decrease number by 1
  write number
```

```
Resultat: 43210
```

for each

...fungerar på två olika sätt. Elementen i en lista eller tal ur ett intervall plockas ett i taget och görs tillgängligt för resten av satsen via en variabel. Satsen upprepas en gång för varje element eller tal.

Exempel:

```
#! /usr/bin/env pseudocode

list equals [“a string”, 4, [“a string inside a list”, 2]]
for each element in list do
  write element, “\n”
```

```
Resultat:
a string
4
[“a string inside a list”, 2]
```

```
#! /usr/bin/env pseudocode

sum equals 0
for each number from 1 to 5 do
  increase sum by number
write sum
```

```
Resultat: 15
```

2.3.9 Indexering

För att plocka ut enskilda element ur strängar och listor går det välja ett specifikt index ur objektet. Indexeringen böjar räkna på 1 och följer engelskans uppräkningsord.

Exempel:

```
>> 1st of [5,2,67,8]
5
>> 2nd of [5,2,67,8]
2
>> 3rd of [5,2,67,8]
67
>> 4th of “this is a string”
“s”
>> 14th of “this is a string”
“i”
>> last of [5,2,67,8]
8
```

2.3.10 Funktioner

En funktion är en sats som gör det möjligt att köra dess tillhörande rader från andra ställen i programmet. En funktion deklareras med ett namn och godtyckligt antal parametrar. Liksom variabler måste funktionsnamn och parametrar bestå av stora och/eller små bokstäver från a till z. Vid anrop till en funktion måste lika många argument som deklarerade parametrar anges. En funktion kan returnera ett värde till platsen för anropet. Funktioner kan inte deklareras inuti en annan sats.

Exempel:

```
#! /usr/bin/env pseudocode

myFunction does
  write “Hello world!”

anotherFunction with firstParameter, secondParameter does
  return firstParameter plus secondParameter

do myFunction
write “\n”, do anotherFunction with 4, 2
```

```
Resultat:
Hello world!
6
```

3. Systemdokumentation

3.1 Allmänt

Pseudocode är ett interpreterat språk implementerat i *Ruby* med hjälp av den givna parsern[1] och är avsett för operativsystem i POSIX-familjen. Det finns flera sätt att köra språket på, nämligen genom att anropa interpretatorn i en terminal:

- **med ett filnamn som argument:**
  Programmet exekveras då med filen text som källkod.
- **med källkoden i inputbufferten (stdin):**
  Programmet exekveras med texten i inputbuffertern som källkod.
- **utan argument och skriva till den rad för rad:**
  Programmet startar då en interaktiv prompt där användaren kör kommandon rad för rad. Detta är det enda läget som för tillfället inte klarar satser över flera rader.

3.2 Installation och systemkrav

Pseudocode har bara testats i Linux men fungerar förmodligen även i andra POSIX-system. För att kunna köra det krävs en installation av Ruby[2] (version 1.9.3 och 2.0 har testats och fungerar).

För att installera *Pseudocode*:

1. Packa upp filen *pseudocode.tar.gz* i valfri mapp
2. Kör filen *install* som administratör.
3. Starta interpretatorn genom att skriva “pseudocode” i terminalen.

3.3 Implementation

3.3.1 Lexikalisk analys

När källkoden läses in till programmet så tas bort kommenterad kod bort först, eftersom den är ointressant för parsern. Efter det så tas alla tecken mellan citationstecken och sparar dessa som Ruby-strängar. Alla tal sparas också som Ruby-integers och Ruby-floats, samt vissa specialuttryck matchas.

När lexern hittar radbrytningar så genereras speciella tokens som behövs för att parsern ska kunna tolka indentering på rätt sätt. Alla dubbletter av radbrytningar ignoreras och representeras av en enda radbrytning vid parsning. Till sist kastas alla mellanslag bort.

[1] Bilaga D
[2] https://www.ruby-lang.org/en/installation/

3.3.2 Parsning

Parsningen går ut på att matcha tokens från lexern mot regler som motsvarar språkets grammatik och därefter skapa Ruby-objekt som motsvarar språkets konstruktioner. Alla matchningar vid parsning skapar antingen en ren Ruby-datatyp (representationen av *Pseudocodes* strängar, tal och boolska värden) eller en nod (egen implementation av Ruby-klasser) som går att evaluera. Alla objekt hamnar till sist i en lista där de är redo att evalueras i tur och ordning.

3.3.3 Evaluering

När parsningen är klar finns alla noder (en nod per sats) i den yttersta noden. Den evaluerar alla noder som den innehåller i tur och ordning genom att kalla på varje nods *evaluate*-funktion. Varje nod har sin egen *evaluate* och varje evaluering resulterar antingen i att det utförs en beräkning, eller att den i sin tur evaluerar de noder som den själv innehåller. Vi har också utökat Rubys inbyggda klasser för att inte behöva evaluera Rubys datatyper annorlunda jämfört med *Pseudocodes* noder.

3.3.4 Nodstruktur

Noderna[3] i *Pseudocode* är implementerade som Ruby-klasser. Alla typer av satser, uttryck och vissa datatyper har en egen nod. De ärver från klassen *SuperNode* som har en klassvariabel som innehåller programmets variabler och funktioner, kallat *scope*. Noder som motsvarar villkorssatser, upprepningssatser och funktioner har sitt eget scope för att hantera lokala variabler. Scope är implementerat dynamiskt, som en klass som känner till sin förälder (ett steg uppåt i kedjan av scopes). Den har funktioner för att sätta och hämta variablers värden samt deklarera och anropa funktioner.

[3] Bilaga C

4. Reflektion

Slutresultatet av vår implementation blev i stort sett som vi ville enligt specifikationen. Vissa detaljer, till exempel typkonvertering, har vi ännu inte implementerat och vissa saker har vi ändrat på gentemot specifikation, till exempel var tanken från början att listor skulle deklareras med nyckelordet “holds” istället för “equals” som andra variabler. Alla variabler, oavsett typ, deklareras nu istället på samma sätt.

När grammatiken var klar började vi med att överföra den, med viss modifikation, till parserregler och började därefter att skriva testfall med kod för det som vi ville implementera. Sedan fortsatte arbetet testdrivet genom hela processen och det har visat sig fungera mycket bra för oss eftersom vi enkelt kunde köra alla testfall och försäkra oss om att förändringarna inte har förstört något annat vi implementerat.

De svåraste delarna av implementationen, som vi haft mest problem med, var alla relaterade till funktioner. Dels hade vi problem med scope, att vi kunde råka evaluera noder i fel scope, där variabler var satta till andra värden och då få fel resultat. Vi hade förväntat oss att det skulle vara svårare att implementera scope än det faktiskt var, även om vi hade vissa problem med det.

Returvärden hade vi också en del problem med, då vi var tvungna att se till att de evaluerades innan de returnerades (vilket också var relaterat till scopes), samt att de avslutade programmet eller funktionen, och inte fortsatte. Liknande problem hade vi med noder som evaluerades flera gånger (satser i funktioner, villkorssatser etc.), att nodens instansvariabler användes istället för lokala variabler, vilket ledde till felaktiga resultat.

Eftersom vi valde att tvinga indentering i språkets syntax så blev vi tvungna att skriva om parsern och lexern en del, vilket gav en ökad förståelse för hur de fungerar. Från början hade det varit bra att få tillgång till någon slags dokumentation för parsern, då vi hade mycket problem med att parsningen tog väldigt lång tid. Problemet berodde på att vi hade lagt vissa tokenmatchningar i “fel” ordning, något vi inte kom underfund med förrän relativt sent.