Elias Ericson(elier502@student.liu.se)
Gustav Lättman (gusla889@student.liu.se)

Tekniska högskolan vid Linköpings universitet 2011-05-25
Innovativ Programmering
TDP019 Projekt: Datorspråk

Innehållsförteckning

Introduktion

Den här rapporten är en del av kursen TDP019 från programmet Innovativ Programmering på Linköpings Tekniska Universitet. Målet med kursen är att skapa ett fullständigt dokummenterat programmeringsspråk. Denna rapport beskriver och förklarar hur språket AutoSystem fungerar.

Snabb överblick

Vårat språk heter AutoSystem. *Auto* betyder *"någonting är automatiserat"*. S*ystem* antyder på operativsystemet. AutoSystem är alltså ett språk för att automatisera vissa Windowsoperativsystem. Vi ville göra något mer än ett vanligt programmeringsspråk, något som urskilde sig från de tidigare språk som vi läst under första året på programmet Innovativ Programmering. Python och Ruby är de två språk som vi inspirerats av mest för uppbyggnaden av själva grundspråksstrukturen. AutoSystem innehåller dom viktigaste byggstenarna för ett modernt programmeringsspråk. Man bör dock notera att AutoSystem inte har stöd för klasser, något som vi ansåg var onödigt för programmeringsspråkets huvudsakliga syfte; simulera musrörelser och knapptryckningar.

Implementeringsplan

Vi hade som plan att först ta fram en nästan helt fungerande grammatik samtidigt som vi läste på om rdparsern, för att lättare kunna använda den senare i implementationen. Även fast vi under föregående kurs TDP007 jobbat en del med det kodstycket så tyckte vi att det kunde vara bra med att förnya och fördjupa våran kunskap inom det. Implementationssteget där koden för själva språket skulle skrivas hade vi tänkt att jobba med samtidigt som vi allt eftersom lade till och ibland skrev om mindre delar i grammatiken. Syntaxen bestämde vi oss dock för ganska tidigt.

Val av verktyg

Vi har valt att använda AutoIT. Det är ett språk för att simulera musrörelser, knapptryckningar och andra finurliga simuleringar på operativsystemet. Det finns ett bibliotek till Ruby som har stöd för att köra AutoIT-kod, vilket gör det lättare för oss att implementera till AutoSystem.

Användarhandledning

Installation

Ruby

För att köra AutoSystemspråket så behöver man Ruby installerat. Lättast är att besöka *Rubys hemsida*[1] och hämta hem den senaste stabila rubyinstallationen för Windows.

AutoIT

För att kunna köra AutoIT-kod direkt i Ruby så behöver man hämta hem ett Rubybiblioteket för AutoIT[2]. Denna fil finner du bland vår källkod. Om du har för avsikt att använda vår källkod för eget bruk så rekommenderar vi att registrera[3] .dll-filen i systemet.

Filstruktur

Språket innehåller fem viktiga filer: source.as, rdparser.rb, autosystem_source.rb och run.rb. För att kunna köra AutoSystem så behövs samtliga filer.

source.as

Det är i denna fil som källkoden för programmerarens program skrivs, som sedan AutoSystems systemfiler tolkar.

rdparser.rb

Vi har valt att använda den redan färdiga ruby-parsern som vi fick tillgång till i början av projektet. autosystem_source.rb använder denna fil för att parsra programmerarens källkod.

autosystem_source.rb

Denna är filen där större delen av vår källkod för projektet ligger. Programmerarens källkod skickas hit efter att run.rb körts.

run.rb

Det är run.rb som man först startar när man vill köra kod i AutoSystem. Man kan se denna fil som en första länk i AutoSystems tolkkedja.

[1] http://rubyforge.org/frs/download.php/74298/rubyinstaller-1.9.2-p180.exe
[2] http://rubygems.org/downloads/au3-0.1.2.gem
[3] Öppna cmd och skriv: regsvr32 [**path**]/AutoItX3.dll (som administrator i vista/windows7)

Programmera i AutoSystem

AutoIT

AutoIT är från början ett programmeringspråk till windows för diverse hantering av operativsystemet. Detta bibliotek är bundet till ruby och det är vad vi använder oss av för att få dess funktionaltitet till standardbiblioteket.

Variabler och tilldelning

Användaren kan tilldela värden till variabler. En variabel har en typ. En typ kan vara en av dessa: integer, float, string, list, bool. Den kan även tilldelas returvärdet av function_call, file, logical-expression, relation-expression och aritmetical-expression.

Snabbdefinition på typerna:

Integer är samma sak som heltal.

Kodexempel:
```
int_variable = 123
```

Float är flyttal, d.v.s. ett tal som har decimaler.

Kodexempel:
```
my_float = 123.2345
```

String är en sträng av tecken, en rad med text.

Kodexempel:
```
str_var = "Hello"
```

List är en lista med element, ett element kan vara alla av dessa variabel typer här.

Kodexempel:
```
my_list = [0,"hej",5.5,[5,8,1],1,34784]
```

Bool står för boolean och hanterar bara två värden: true, false (ett och noll).

Kodexempel:
```
proceed1 = true
proceed2 = 10 < 2
```

Function_call kör kod från en specificerad funktion.

Kodexempel:
```
myfunc = calc(10,5)
```

File är en klass med ett fåtal funktioner för skriv och läsning till filer.

Kodexempel:
```
a = File.new("data.txt", "w")
```

Operatorer

Logiska uttryck

Logiska expression använder: and, or och not. Med dessa jämför man flera predikatuttryck för att slutligen får ett sanningsvärde.

```
Kodexempel:
if true and true
```

Lista över logiska operatorer

Operator	Beskrivning
and	Och. Tar två expressions och kontrollerar om båda är sanna. returnerar true
or	Eller. Tar två expressions och kontrollerar om ena är sann. returnerar true
not	Inte. inverterar retur värdet. Tex om man vill checka om ett expression är falskt. i stället för expression == false kan man skriva not expression.

Relationsuttryck

Relations expression använder relationsoperatorer (se lista nedan). Med dessa jämför man flera relationsuttryck för att slutligen få ett sanningsvärde.

```
Kodexempel:
expr = 5==1
```

Lista över relationsoperatorer

Operator	Beskrivning
<	Mindre än
>	Större än
==	Lika med
>=	Större eller lika med
<=	Mmindre eller lika med
!=	Inte lika med

Aritmetiska uttryck

Aritmetiska expressions hanterar addition, substraktion, divition, multiplikation och modul. Kan även hantera parenteser för att få prioritet på beräkningen.

```
Kodexempel:
expr = ((50+10)/2)-(132%2)
expr += 2
```

Lista över aritmetiska operatorer

Operator	Beskrivning
+	Adderar två tal eller strängar. Om man adderar en tal med en sträng blir det en sträng.
-	Subtraherar två tal. Om man skriver - framför en sträng blir den inverterad.
*	Multiplicerar två tal eller strängar. Om man multiplicerar en sträng med ett tal, repeteras strängen så många gånger.
/	Dividerar två tal. Om man dividerar en sträng med ett tal så divideras strängens längd på talet och resultatet blir att strängen blir så lång.
%	Modyl på två tal. Liknar dividering men man får rästen från divideringen istället.

Repetitions-satsen

Rep-satsen står för engelskans repeat, och hanterar loopar i språket. En loop innebär att en eller flera rader kod upprepas så många gånger man har specificerat. Om man jämför med andra programmeringsspråk är rep-satsen en blandning av en for-loop och en while-loop i AutoSystem. Man kan definiera rep-satsen på tre olika sätt:

1. Första typen man definiera den på har två värden, t.ex. 2 och 10. Dessa två värden skapar ett intervall som man därefter itererar över. Man definierar också ett variabelnamn. Variablen kommer starta på värdet 2 och sluta på värdet 9 i detta fall. Variabeln ökar med ett för varje intervall.

   ```
   Kodexempel:
   rep i in 2,10
        < i
   end_rep
   ```

   Här definieras variabeln i som kommer initieras med värdet 2 och sluta på 9. Koden som kommer repetaras i detta fall är **"< i"** vilket kommer skriva ut värdet på variabeln i. Det kommer reslutera i en utskrift med alla värden från två till nio.

2. Andra typen fungerar på nästan exakt samma sätt förutom att man använder en lista att repetera över istället för ett intervall. Variabeln som man också definierar kommer istället för att börja på t.ex. 2 som i förra exemplet, börja på första värdet som listan innehåller. Stegvis kommer den ersätta sitt värde med kommande värde. Denna repetitionssats körs så många gånger som listan har element.

```
Kodexempel:
rep item in my_list
     < item
end_rep
```

Här måste man ha definierat variabeln my_list sen tidigare denna lista kan innehålla vad som helst. "< item" kommer skriva ut värdet på elementen från my_list.

3. Tredje typen är ganska olik därför att här kan man använda ett slutvilkor. Man kan använda det hur man vill, och om expression returnerar true kommer den fortsätta repetitionen. Repetitionen pågår så länge slutvilkoret är sant. En repetition kan avslutas med break.

```
Kodexempel:
num = 0
rep num<100
     < num
     num+=10
end_rep
```

Här initierar användaren en variabel: *num* innan rep-satsen. I rep-satsen används ett expression som returnerar true, om variabeln num är mindre än 100. Koden: "< num" skriver ut värdet på num, och num+=10 ökar num med 10.

```
Kodexempel:
num = 0
rep true
     num+=1
     if num == 10
          break
     end_if
end_rep
```

Här returneras true direkt i rep-satsen vilket skapar en oändlig loop. För att den ska sluta kan man använda sig av kommandot break.

Villkorssats

Det if-satsen gör är att ta ett eller flera expressions, jämför dom och ser om allt returnerar sant. Om så är fallet går programmet vidare till koden under if-satsen. Annars utesluts if-satsens kodkropp och koden efter *end_if* exekveras istället.

```
Kodexempel:
i = 5
if i < 10
     < i
end_if
< "End"
```

If-satsen kan hantera oändligt många kontrolleringar. Man kan också använda sig av parenteser för att få prioritet på vad som först ska kontrolleras. Dessutom kan man ha andra if-satser i if-satser (rekursivitet).

```
Kodexempel:
if ((i>=2 and 5>2) and (5!=2 and 10 == h)) and i == 2)
     if i > 10
          < "it is true"
     end_if
     yes = 10
end_if
```

Funktioner

Eftersom vi använder oss av det externa biblioteket AutoIt[4] så gör det att vi har en hel del funktioner att hålla reda på. Vi gör detta genom att särskilja på de funktioner som programmeraren själv definierar och de funktioner som följer med och redan är inbyggt i språket. Men vad skulle hända om programmeraren definierar en funktion av samma namn som en från våra inbyggda funktioner? Vi har valt att låta programmeraren skriva över en redan befintlig fördefinierad funktion. Det kändes mest naturligt att göra på det sättet.

Definiera en egen funktion

AutoSystem är till största delen byggt för att procedurell programmering ska gå så snabbt och enkelt som möjligt, det ska vara lätt att definiera nya funktioner. En funktion i AutoSystem skulle t.ex. kunna vara en procedur för att röra musen i ett speciellt mönster en viss tid. Eller att låta tangentbordet skriva in en text för att sedan musen i sin tur ska röra sig till en viss punkt på skärmen och klicka. En funktionell funktionsdefinition skulle kunna se ut så här:

```
Kodexempel:
def function()
       < "Hello World"
end_def
```

[4] Bibliotek är en samling av funktioner. Som en bokhylla med en massa olika böcker.

Med denna kod så har vi definierat en funktion "function" som kommer att sparas undan till ett senare tillfälle. Syntaxen def är nödvändig för att AutoSystem ska förstå att det är en funktionsdefinition som det handlar om. Efter funktionssyntaxen så är det funktionsnamnet som måste namnges följt av vänsterparantes, parameterlistan och högerparantes.
För att definiera en funktion med parametrar så skriver man så här:

```
Kodexempel:
def paramFunc(param1,_param2)
        return (param1 +_param2)
end_def
```

En enkel funktion som tar in två parametrar: _param1 och _param2. Dessa två parametrar adderas och funktionen retunerar sedan summan. En funktion kan alltså definieras med eller utan parametrar. Tillåtet antal parametrar är nästan obegränsat många. Konstruktionen return skickar tillbaka det värde som står efter på samma rad och bryter sig ur det aktuella kodblocket.

Funktionsanrop

För att kalla på en funktion anger man funktionsnamn följt av vänster- och högerparantes. När man gjort det kommer AutoSystem att köra funktionens fördefinierade funktionskropp och i vissa fall retunera ett värde.

```
Kodexempel:
function()
```

För att kalla på en funktion med parametrar skriver man på följande sätt:

```
Kodexempel:
sum = paramFunc(10, 50)
```

Funktionen paramFunc retunerar summan av de två argumenten vilket gör att variabeln sum kommer att tilldelas värdet 60, i just detta fall.
Anrop av en fördefinierad simuleringsfunktion skrivs på samma sätt. Givetvis måste funktionsnamnet skrivas exakt så som funktionen i funktionslistan[5] över tillgängliga fördefinierade funktioner ser ut.

[5] Se inbyggda funktioner.

Räckvidd

Variabler och funktioner

Att nå en variabel eller funktion som inte är deklarerad inne i det aktiva blocket är tillåtet. Man kan alltså nå variabler utanför det block man befinner sig i.

Skuggning

Med hjälp av skuggning så kan man skapa variabler i djupare kodblock utan att skriva över variabler som befinner sig högre upp. Detta kan vara användbart till t.ex. repetitionssatser. I kodexemplet nedan så illustreras detta fenomen. Det värde som skrivs ut är värdet på det "närmaste" x, d.v.s. 0, 1 och 2.

```
Kodexempel:
x = 30
     rep x in 0,2
          < x
     end_rep
=>0
=>1
=>2
```

Standardbiblioteket

Standardbiblioteket bygger bl.a. på att man ska kunna använda alla funktioner från ett AutoItbibliotek som vi laddar in i Ruby. Det ska också kunna ladda och läsa filer från hårddisken. Dessa två typer av standardbibliotek är uppdelade. Filhanteringen matchas direkt i parsern medan AutoItfunktionerna hanteras som strängar som sedan evalueras som Rubykod direkt.

Inbyggda Funktioner

Funktion	Beskrivning
File(filename,mode)	Laddar in en fil till minnet. **filename** är platsen den är eller ska skapas på (en string), **mode** är hur den ska hanteras: **r** (enbart läsning), **r+** (läsning och skrivning), **w** (enbart skrivning och skapar/ersätter filen), **w+** (läsning och skrivning, skapar/ersätter filen), **a** (enbart skrivning, om filen finns fortsätter man skriva i slutet på filen annars om den inte finns skapas en ny fil), **a+** (läsning och skrivning, om filen finns fortsätter man skiva i slutet annars om den inte finns skapas en ny fil), mode skrivs som en string.
read()	En subfunktion till File som läser in all data från filen till en lista
write(data)	En subfunktion till File som skriver data. Data kan vara tex en string, en integer, en float etc.
close()	En subfunktion till File som stänger skriv och/eller läsning från minnet.
hex(int)	Konverterar heltal(integer) till ett hexadecimalt nummer.
int(hex)	Konverterar ett hexadecimalt nummer till ett heltal(integer).

Inbyggda AutoItfunktioner

Funktion	Beskrivning
MouseClick(“button”, x, y, clicks, speed)	Simulerar en musklickning med diverse alternativ. Button kan T.ex. vara “left”.
MouseClickDrag(“button”, x1, y1, x2, y2, speed)	Simulerar en markering med musen med diverse alternativ.
MouseDown(“button”)	Trycker ner en knapp på musen
MouseGetCursor()	Returnerar muspekarens ID nummer.
MouseGetPosX()	Returnerar musens x position.
MouseGetPosY()	Returnerar musens y position.
MouseMove(x, y, speed)	Flyttar musen.
MouseUp()	Släpper en knapp som är nere på musen.
MouseWheel(“direction”, clicks)	Genomför en skrollning med musen. Direction kan vara “up” eller “down”.
PixelChecksum(left, top, right, bottom , step , hwnd, mode)	Generar en så kallad “checksum” för en region av pixlar.
PixelGetColor(x, y, hwnd)	Returnerar färgen på en position.
PixelSearch(left, top, right, bottom, color, shade-variation, step, hwnd)	Söker i en region efter en färg på en pixel.
ProcessClose("process")	Stänger en specificerad(ID) process.
ProcessExists(“process”)	Checkar om den specificerade processen existerar.
ProcessSetPriority("process", priority)	Ändrar prioritet för den specificerade processen.
ProcessWait("process", timeout)	Pausar körningen av koden tills det att processen existerar.
ProcessWaitClose("process", timeout)	Pausar körningen av koden tills det att processen inte existerar.
RegDeleteKey("keyname")	Tar bort en nyckel i registret.
RegDeleteVal("keyname", value)	Tar bort ett värde från en nyckel i registret.
RegEnumKey("keyname", instance)	Läser namnet på en undernyckel beroende på sin instans.

RegEnumVal("keyname", instance)	Läser namnet på ett värde beroende på sin instans.
RegRead("keyname", "valuename”)	Läser ett värde från registret.
RegWrite("keyname", "valuename", "type", value)	Skapar en nyckel eller ett värde i registret.
Run("program", "workingdir", show_flag)	Kör ett externt program.
RunAsSet(user, domain, password, options)	Kör ett externt program med en annan användare.
RunWait(run, wait, show flags)	Kör ett externt program och pausar skriptet tills att programmet är avslutat.
Send("keys", flag)	Skickar simulerade knapptryckningar till det aktiverade fönstret.
Shutdown(code, reason)	Stänger av datorn.
Sleep(delay)	Pausar skriptet/exekveringen.
StatusbarGetText("title", "text", part)	Retrieves the text from a standard status bar control. Får texten från en standard “status bar” kontrol.
ToolTip("text" , x , y)	Skapar en “tooltip”-ruta på skären.
WinActivate(strTitle, strText)	Aktiverar/fokuserar ett fönster.
WinActive(strTitle, strText)	Checkar om ett specificerat fönster existerar och är förtillfället aktivt.
WinClose(strTitle, strText)	Stänger ett fönster.
WinExists(strTitle, strText)	Checkar om ett specificerat fönster existerar.
WinGetCaretPosX()	Returnerar kordinaten x av fönstret.
WinGetCaretPosY()	Returnerar kordinaten y av fönstret.
WinGetClassList(strTitle, strText)	Får alla klasserna från ett fönster.
WinGetClientSizeHeight(strTitle, strText)	Får höjden på ett givet fönster.
WinGetClientSizeWidth(strTitle, strText)	Får brädden på ett givet fönster.
WinGetHandle(strTitle, strText)	Får det interna värdet från ett fönster.
WinGetPosX(strTitle, strText)	Får position x från ett givet fönster.

WinGetPosY(strTitle, strText)	Får position y från ett givet fönster.
WinGetPosHeight(strTitle, strText)	Får höjden av ett givet fönster.
WinGetPosWidth(strTitle, strText)	Får brädden av ett givet fönster.
WinGetProcess(strTitle, strText)	Får processens ID (PID) som är associerat med ett fönster.
WinGetState(strTitle, strText)	Får ett givet fönsters status.
WinGetText(strTitle, strText)	Får texten från ett fönster.
WinGetTitle(strTitle, strText)	Får ett fönsters fulla titel.
WinKill(strTitle, strText)	Tvinga att ett fönster stängs.
WinList(strTitle, strText)	Får en lista av fönster.
WinMenuSelectItem(strTitle, strText, strItem1, strItem2, strItem3, strItem4, strItem5, strItem6, strItem7, strItem8)	Anropa ett meny objekt från ett fönster.
WinMinimizeAll()	Minimerar alla fönster.
WinMinimizeAllUndo()	Ångrar en föredetta WinMinimizeAll funktion.
WinMove(strTitle, strText, nX, nY, nWidth, nHeight)	Flyttar och/eller ändrar storlek på ett fönster
WinSetOnTop(title, text, flag)	Ändra attributen “Alltid överst” på ett fönster.
WinSetState(title, text, flags)	Visa, göm, minimera, maximera, återställ ett fönster.
WinSetTitle(title, text, newTitle)	Ändrar ett fönsters titel.
WinSetTrans(title, text, trans)	Ändrar ett fönsters genomskinlighet (Windows 2000/XP or later).
WinWait(title, text, timeout)	Pausar exekvering av skriptet tills det att det anropade fönstret existerar.
WinWaitActive(title, text, timeout)	Pausar exekvering av skriptet tills det att det anropade fönstret är aktivt.
WinWaitClose(title, text, timeout)	Pausar exekvering av skriptet tills det att det anropade fönstret är stängt.
WinWaitNotActive(title, text, timeout)	Pausar exekvering av skriptet tills det att det anropade fönstret inte är aktivt.
IsAdmin()	Kollar om användaren har administrationsrättigheter.

IniDelete(filename, section, key)	Tar bort ett värde från en standard .ini-fil.
IniRead(filename, section, key, default)	Läser ett värde från en standard .ini-fil.
IniWrite(filename, section, key, value)	Skriver ett värde till en standard .ini-fil.
BlockInput(flag)	Aktivera/avaktivera musen och tangentbordet.
CDTray(drive, action)	Öppnar eller stänger CD-läsaren.
ClipGet()	Hämta text från urklippet.
ClipPut(strClip)	Skriver en text till urklippet.

Programexempel

Det här mycket enkla programexempel flyttar musen till mitten av skärmen, som i detta fall är fördefinierat som 800x600. Därefter genomförs en serie av musflyttningar och klick för att skapa en tom mapp.

Kodexempel:

```
def mouseToCenter()
      screen_res_x = 800 / 2
      screen_res_y = 600 / 2

      MouseMove(screen_res_x, screen_res_y)
end_def

def createNewFolder()
      MouseClick("right")
      x = MouseGetPosX()
      y = MouseGetPosY()
      MouseMove((x + 100), (y + 195))

      Sleep(1000)

      MouseMove((x + 400), (y + 195))
      MouseClick()
end_def


mouseToCenter()
createNewFolder()
```

Systemdokumentation

Lexern

Vi använder oss av en parser; recursive decent parser kallad "rdparser". Denna fil innehåller två klasser: Rule och Parser. Det är Ruleklassen vi använder för att plocka ut så kallade tokens. En token är en grupp bokstäver tagna från en sträng. Dom bokstäver som blir valda beror på syntaxen man har bestämt.

Tokens vi plockar ut:
String, vi kör ett reguljärt uttryck som matchar strängar: det ska börja med tecknet " följt av vilket tecken som helst förutom tecknet " slutligen matchas tecknet " igen för att sluta strängen.

På liknande sätt som string beskrevs, matchas alla andra delar för att få ut alla tokens. Vi plockar ut:

- Ny rad
- Jämförelse operatorer t.ex.: ==, >, <
- Paranteser: ( )
- Aritmetiska operatorer: + - * /
- Tilldelnings operatorer: = += -=
- Flyttal t.ex: 123.04
- Heltal t.ex: 32452
- Mellanrum och tabbningar tas bort
- Kommentarer tas också bort (om # täcknet kommer så tas all text bakom det bort)

Här nedanför är några rader ur källkoden som visar hur det ser ut när man plockar ut tokens. Här använder man funktionen token som hanterar reguljära uttryck. Det som står inom klammerparenteser skickas vidare till parsern. I detta fall ett "m". Om det inte skulle vara någonting där skulle man "kasta bort" det man läser in. Det är så man gör med bl.a. kommentarer och mellanrum.

Kodexempel:

```
#CompareOP
token(/==/) {|m| m}
token(/!=/) {|m| m}
token(/</) {|m| m}
token(/>/) {|m| m}
token(/<=/) {|m| m}
token(/>=/) {|m| m}
```

Parsern

Som nämdes tidigare så används Rdparser. Med hälp av detta tillägg kan man precis som med tokens skriva regler för hur programmet ska agera. En regel innehåller en eller flera matchningar. En match är lik en token, men skillnaden är att reglerna matchar med hjälp av tokens.

Kodexempel:

```
rule :break do
  match(“break”) {|break_stmt| BreakStatement.new()}
end
```

I detta exempel ser vi regeln break som matchar en sträng “break”. Regeln skapar en nod BreakStatement som skickas vidare uppåt.

Noder

Programmet körs med hjälp av ett abstrakt syntaxträd. Det använder sig av noder. Noder skapas från parsern i form av klasser. En nod-klass är uppbyggd med två huvudfunktioner:
1. En initieringsfunktion som sparar ner om det har skickats variabler från matchningen. Det är matchningen som initierar noden och därmed kan skicka in parametrar.
2. En evalueringsfunktion som gör vissa beräkningar och oftast returnerar ett värde.

Kodexempel:

```
class ListDefinition
  def initialize(elements)
    @elements = elements
  end

  def evaluate()
    if @elements!=nil
    return @elements.evaluate()
    else
    return []
    end
  end
end
```

I det här exemplet ser man klassen ListDefinition. Här kan man tydligt se dessa två huvudklasser; initialize och evaluate. Initialize har ett argument som skapar en instansvariabel variabel; @elements. Nu kan man i funktionen evaluate använda denna variabel. I detta fall kontrolleras om @elements inte är nil eftersom det är en rubylista vi vill skapa. Om så är fallet returneras evalueringen, annars returneras en tom lista. Här är själva matchningen som skapar noden och skickar in parametern:

Kodexempel:

```
rule :list_def do
  match("[", :elements, "]") {|_,elements,_|
ListDefinition.new(elements)}
  match("[", "]") {|_,_| ListDefinition.new(nil)}
```

Här krävs det två matchningar eftersom man kan definiera en lista med eller utan element.

Program vi använt

Vi har programmerat i Gedit och Eclipse med hjälp av ett Rubyaddon. Vi har även använt oss av kommandotolken och terminalen för att snabbtesta Rubykod.

För en enkel och smidig fildelning så har vi använt oss av Dropbox som är ett program där man kan skapa delade mappar och filer över internet.

Räckvidd

Variabler och funktioner

Variabel- och funktionsräckvidden i AutoSystem sträcker sig från grundnivån 0, till den djupast funktionen/repetitionssatsen. När man deklarerar en variabel på grundnivån så kommer den att tilldelas specialvärdet 0 för att veta vilken nivå den tillhör. När man deklarerar en variabel inne i en funktionskropp som i sin tur har definierats på grundnivån så får denna variabel specialvärdet 1. Specialvärdet har inget att göra med variabelns egentliga värde utan används enbart "under huven" för att hålla reda på till vilken nivå variabeln i fråga hör hemma. Metoden vi använder oss av kallas för dynamisk räckvidd. När man letar efter en variabel eller funktion så letar man stegvis bakåt i historiken över där man tidigare har befunnit sig.

Skuggning

Skuggning av variabler kan vara väldigt användbart speciellt när det kommer till användning av nästlade repetitionssatser. Ta en titt och fundera över följande kod:

```
Kodexempel:
x = 64
rep x in 0,10
     x = 1000
     rep y in 2,15
          < x
     end_rep
end_rep
< x
```

Vid det första utskriftstillfället i den andra repetitionssatsen, vilket värde var det som skrevs ut? Eftersom AutoSystem använder sig av dynamisk räckvidd så kommer utskriften att bli 1000. Om man inte hade tilldelat *x* värdet 1000 i den första repetitionssatsen så skulle utskriften i den andra repetitionssatsen blivit 0, följt av 1, 2, 3 osv. Däremot påverkas inte värdet av *x* på grundnivån vilket gör att vid det sista utskriftstillfället så kommer *x* fortfarande vara 64.

Grammatik

program:= stmt_list

comment:= (/#*.**#/)

stmt_list:= stmt_list :stmt_breaker
stmt_breaker

stmt_breaker:= simple_stmt :NEWLINE
comment

simple_stmt:= file_operations
assign_stmt
func_def
func_call
quick_assign_stmt
break
if_stmt
rep_stmt
return
print_i
print_o
expr

return:= "return" expr
"return"

break:= “break”

print_o:= "<" expr
"<" string

print_i:= identifier "<"

parameters:= parameters "," parameter
parameter

parameter:= identifier

func_def:= "def" identifier "("parameters ")" NEWLINE body "end_def"
"def" identifier "(" ")" NEWLINE body "end_def"

```
func_call:=      identifier "(" arguments ")"
                 identifier "(" ")"

arguments:=      arguments "," argument
                 argument

argument:=       expr
                 func_call

if_stmt:=        "if" expr NEWLINE body "end_if"

rep_stmt:=       "rep" identifier "in" integer "," integer
                 "rep" identifier "in" variable_call NEWLINE body "end_rep"
                 "rep" identifier "in" list_def NEWLINE body "end_rep"
                 "rep" expr NEWLINE body "end_rep"

list_def:=       "[" elements "]"
                 "[" "]"

elements:=       elements "," element
                 element

element:=        expr
                 list_def
                 func_call

list_call:=      identifier "[" list_brackets "]"

list_brackets:=  list_brackets list_bracket
                 list_bracket

list_bracket:=   "[" variable_call "]"
                 "[" integer "]"
NEWLINE:=        "\n"

assign_stmt:=    identifier "=" func_call
                 identifier "=" list_call
                identifier "=" file_new
                 identifier "=" file_read
                identifier "=" list_def
                identifier "=" expr
```

```
quick_assign_stmt:=     identifier "+" calculate_expr
                        identifier "-" calculate_expr
                        identifier "*" calculate_expr
                        identifier "/" calculate_expr
                        identifier "%" calculate_expr

identifier:= /(?!true|false|if|end_if|rep|end_rep|return|in|def|break|end_def|hex|int|File)^([a-zA-Z_][a-zA-Z_\d]*)/

variable_call:=  identifier

body:=           stmt_list

expr:=           or_expr

or_expr:=        and_expr
                 or_expr

and_expr:=       not_expr
                 and_expr

not_expr:=       relation_expr
                 "not" not_expr

relation_expr:=  calculate_expr relation_operator calculate_expr
                 calculate_expr
                 bool
                 variable_call

relation_operator:=  "<"
                     ">"
                     "<="
                     ">="
                     "=="
                     "!="

calculate_expr:=         add_calculate_expr

add_calculate_expr:=     multi_calculate_expr
                         add_calculate_expr "+" multi_calculate_expr
                         add_calculate_expr "-" multi_calculate_expr
```

```
multi_calculate_expr:=  neg_expr
                        multi_calculate_expr "%" multi_calculate_expr
                        multi_calculate_expr "*" multi_calculate_expr
                        multi_calculate_expr "/" multi_calculate_expr

neg_expr:=      atom
                "-" atom

atom:=          paranthese
                integer
                float
                variable_call
                string
                hex_to_int
                int_to_hex

paranthese:=    "(" add_calculate_expr ")"

integer:=       Integer

float:=         Float

string:=        "" /.*/ ""

bool:=          true
                false

true:=          /true/

false:=         /false/

file_new:=       "File" "." "new"  "(", string "," string ")"

file_operations:=       identifier  "."  "write"  "("  string ")"
                        identifier "." "close" "(“ ")"

file_read:=     identifier  "."  "read"  "("  ")"

int_to_hex:=    "hex"  "("  integer  ")"
                "hex" "("  variabel_call  ")"

hex_to_int:=    "int" "(" string ")"
                "int" "(" variabel_call  ")"
```

Reflektion och erfarenheter

Till en början fortsatte vi att lära oss mer av rdparsern som man hade använt från TDP007 och som man helst skulle använda i denna kurs också. Vi tittade på hur koden var strukturerad och hur *tokens* och *matches* fungerade.

Vi satte oss ner och funderade på vad för sorts språk vi ville göra. Vi skissade på idéer och skrev ner hur syntaxen och semantiken skulle se ut. Detta var mycket bra, man kunde på detta sätt gå tillbaka och titta på vad man hade tänkt tidigare etc.

Gramatiken började vi skissa på direkt efter detta utefter vad vi hade pratat om. Vi jobbade i början av projektet på två olika håll vilket ledde till att vi fick göra om några saker. T.ex. så arbetade vi på ett system som byggde på uppdateringar, en funktion som gjorde uträkningar och sedan kallade på sig själv. Innan denna funktion som vi kallade för “run”, laddade vi in källkoden i form av en textfil som sedan blev splittad på alla nya rader. Detta skapade en lista med element, innehållet i elementen blev då raderna och varje rad parsades då i run-funktionen. run-funktionen hanterade current_line som då allt byggde på. T.ex. rep-satsen som uppdaterar kod flera gånger, ändrade värdet på current_line och flyttade upp current_line varje repetition. På detta vis hanterades funktioner också.

Vi fick senare lärdom av att noder var det bästa sättet att hantera språket på i denna kurs. Detta gjorde att vi skrev om våran källkod nästan från början vilket var frustrerande. Resultatet blev en i alla fall mycket mer strukturerat kod. Under projektets gång har vi ändrat och förfinat grammatiken för att den ska bli så optimerad som möjligt. Vi kom även på i efterhand att vi saknade viss funktionalitet i grammatiken som vi ville ha i språket. Men med vår trevliga grammatikgrund så gick det ganska smärtfritt att lägga till. Bättre kommunikation emellan oss två i gruppen men även kursledningen, speciellt så hade mer och bättre information från kursenledningen varit underlättande.

Vi hade lov i ungefär två veckor vilket gjorde att vi kom ur fas i projektet. För när det var dags att sätta igång igen så kändes det som om vi i precis skulle vara klara. Vi effektiviserade den kvarstående tid extra mycket och jobbade intensivt.

Referenser

Win32OLE [www]
http://homepage1.nifty.com/markey/ruby/win32ole/index_e.html
Hämtad 2011-05-12

Ruby [www]
http://www.ruby-lang.org/en/
Hämtad 2011-05-12

Ruby [www]
http://www.ruby-lang.org/en/downloads/
Hämtad 2011-05-12

AutoItlibrary [www] http://robotframework-autoitlibrary.googlecode.com/svn-history/r15/trunk/doc/AutoItLibrary.html
Hämtad 2011-05-12

Rubygems [www]
http://rubygems.org/gems/au3
Hämtad den 2011-05-12