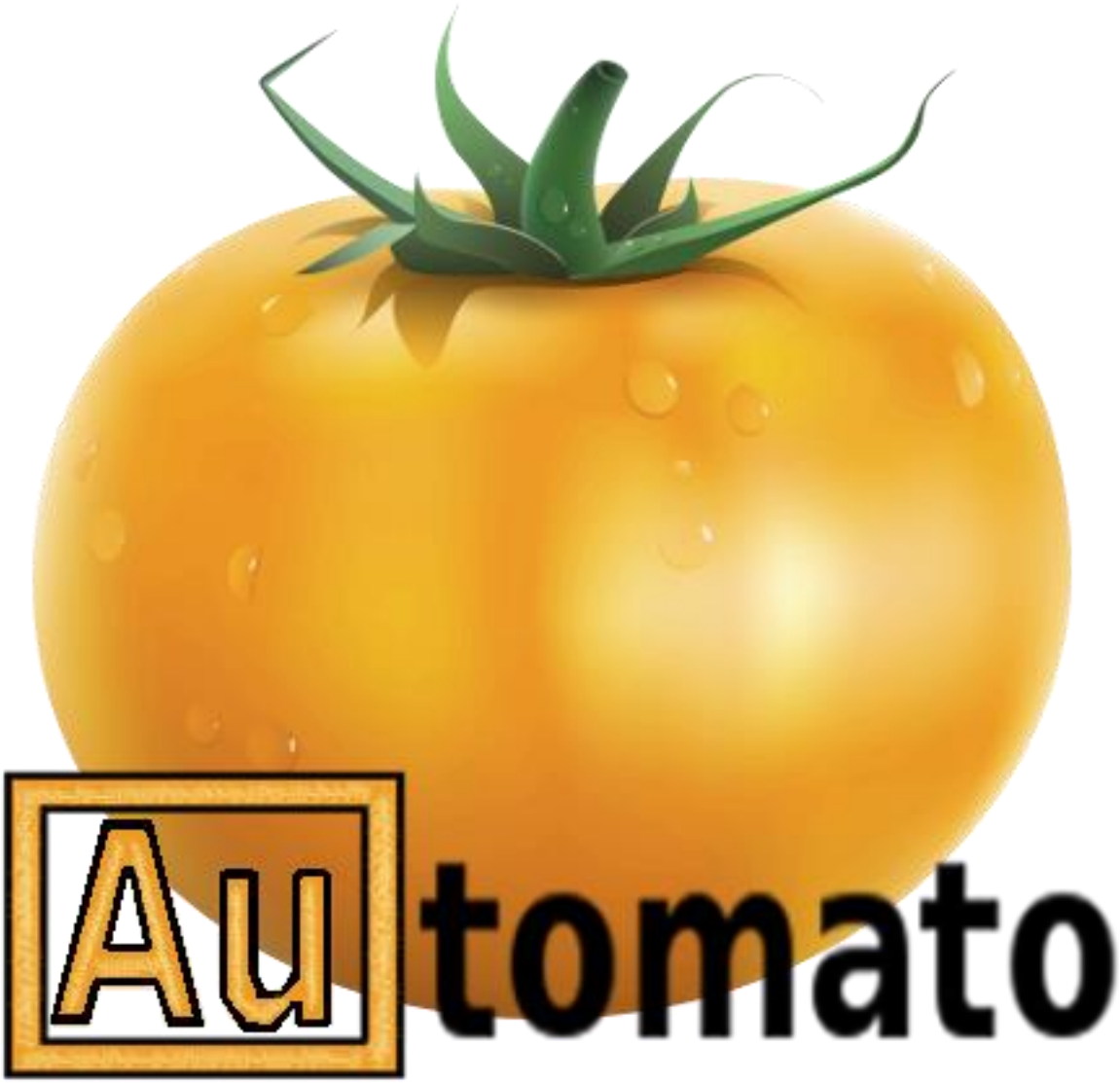


Ett programspråk för automation

Anton Sköld

Gustaf Bodemar

Automato

Ett programspråk för automation

Inledning

Detta är ett projekt som har utförts på IP-programmet, i kursen TDP019 Projekt: Datorspråk. Vi läser denna kurs i den andra terminen på programmet.

Projektet har handlat om att utveckla ett eget datorspråk med ett visst syfte. Vi har valt att skapa ett språk som kan användas för att snabbt och enkelt automatisera mindre sysslor på datorn, dock är språket endas till för användning på Windows OS, då vissa av de använda paketen är Windows beroende.

Språket inkluderar automationsfunktioner, som att: Flytta på muspekaren, klicka med musen (höger/vänster/mushjul), och även att klicka på tangentbordets tangenter. Språket är designat med enkel användning i åtanke.

Denna rapport består av 3 avsnitt, en användarhandledning, en systemdokumentation, samt ett reflektionsavsnitt som beskriver erfarenheter vi fått under projektets gång.

Användarhandledning

Syftet med språket är att enkelt kunna automatisera små sysslor på datorn. Programspråket kan flytta på och klicka med användarens muspekare, och kan även använda användarens tangentbord till att utföra knappkombinationer och skriva text. Språket är designat för att vara lättanvänt och smidigt.

Datatyper

Språket stödjer fem grundläggande datatyper. De är heltal, flyttal (decimaltal), booleska värden (sant/falskt), strängar (vanlig text). Men tillåter även tomta värden ”Nil”. Alla dessa datatyper kan lagras i variabler. Exempel på användning:

```
a = True;
b = False;
c = 123;
d = 5.2;
e = ”Hejsan”;
f = e
g = Nil
```

Koden ovan tilldelar booleska värden till variablerna a och b, heltal och decimaltal till c och d, en sträng ”Hejsan” till variabel e. Variabel f blir tilldelad värdet på variabel e, vilket är strängen ”Hejsan”.

Variabeln g blir tilldelad Nil, vilket är ett tomt värde. Detta värde används främst i mer komplexa språk-funktioner, och tas upp längre ner.

Aritmetiska uträkningar

Språket stödjer aritmetiska uträkningar med heltal och decimaltal, som följer räkneordningen. Det finns även stöd för heltalsdivision och moduloräkning. Exempel nedan:

```
# a blir tilldelad 122.
a = 5 * (3 + 2)^2 – 3;

# b och c blir tilldelade kvoten respektive resten av 7 delat med 2.
b = 7 // 2;
c = 7 % 2
```

I detta fall blir a tilldelad värdet 122, eftersom språket följer räkneordningen. Variabel b blir tilldelad 3, och c blir tilldelad 1.

Variabeloperationer

Språket stödjer operationer som ändrar på redan tilldelade variabler. Exempel:

```
a = 5;          # a tilldelas värdet 5
a += 2;         # a höjs med 2, blir 7.
a -= 3;         # a sänks med 3, blir 4.
a *= 2;         # a multipliceras med 2, blir 8.
a /= 2;         # a divideras med 2, blir 4.
a ^= 2;         # a upphöjs med 2, blir 16.
a %= 7          # a blir resten av en heltalsdivision med 7, blir 2.
a //= 2;        # a divideras med 2, och blir tillsatt kvoten 2.
```

Här kan även parallella operationer användas, exempelvis:

```
a,b += 2,2;     # a och b höjs med 2.
```

Det finns även genvägar för att snabbt ändra på en variabels värde. Exempel:

```
a = 5;          # a tilldelas värdet 5
a++;            # a höjs med 1, blir 6.
a--;            # a sänks med 1, blir 5.
a**;            # a multipliceras med sig själv, blir 25.
a^^             # a höjs upp till sig själv, blir 25^25, vilket är ett väldigt stort tal.
```

Boolesk logik

Språket har stöd för boolesks logik, med ”not”, ”and”, och ”or”-operatörer. Dessa operatörer följer prioritetsordningen not > and > or, där not utvärderas först. Denna ordning kan brytas med parenteser. Exempel:

```
a = True and False;                 # a blir False.
a = True or False;                  # a blir True.
a = not False;                      # a blir True.
a = not False and (True or False);  # a blir True.

b = True;                           # b tilldelas värdet True
c = not b                           # c tilldelas b’s motsatta värde, vilket är False.
```

Jämförelser kan även användas som del av boolesk logik. Exempel:

```
a = 5 > 3;              # a tilldelas True, eftersom 5 är större än 3.
a = 5 < 3;              # a tilldelas False, eftersom 3 inte är större än 5.
a = 5 == 3;             # a tilldelas False, eftersom 5 inte är ekvivalent med 3.
a = 5 != 3;             # a tilldelas True, eftersom 5 är olik 3.

a = 5 > 3 and 5 != 3    # a tilldelas True, efterom 5 är större än 3, och 5 är olik 3.
```

Tilldelning

Den enklaste tilldelningen har vi redan visat exempel på, detta är exempelvis a=2 (a tilldelas 2).

Automato stödjer dock även ”parallella tilldelningar”. Exempel:

```
a, b = 1, 2;      # a tilldelas 1, b tilldelas 2.
a, b += 2, 3      # a och b höjs med 2 respektive 3. a blir 3, b blir 5.
```

Denna tilldelning kan förlängas mer, dock måste båda sidorna vara balanserade för att det ska vara en godkänd tilldelning.

Automato har även support för tilldelning av funktionsresultat. Exempel

```
fun foo () {Return 1}
a= foo()
```

Funktioner kommer förklaras längre ned, men vet bara att den returnerar en sak, 1. a blir då tilldelad 1.

Tilldelningar med funktioner kan kombineras på många olika sätt, det enda kravet är att funktionerna sammanlagt returnerar lika många saker som ska bli tilldelat. Några exempel:

```
fun foo () {Return 1};
a,b = foo(), foo();

fun bar() {Return 1,2};
a,b = bar();

a,b,c = foo(), bar();

a,b,c,d = 1, foo(), bar();

x,y,z,w = 1, a, bar()
```

Scope

Innan vi förklarar de mer avancerade strukturerna i Automato, så ska vi ha en genomgång på scope(räckvidd).

Av standard så har Automato scope aktiverat, på funktioner, men inte på kontrollsatser. Automato är dock skrivet sådant att scope i kontrollsatser kan aktiveras, samt att man kan köra Automato utan scope. I filen classes.rb kan dessa variabler sättas till true eller false:

```
$use_scope = true
$use_scope_if_else = false
$use_scope_loop = false
$use_scope_functions = true
```

$use_scope dikterar över all scope-funktionalitet, om den är false, är alla scope avaktiverade.

Automato har en liten skillnad jämfört med andra språk som Ruby eller Python, i det att en funktion (eller annan struktur som har scope) kan använda variabler ifrån det scope den blev kallad ifrån.

Dock så kan variabler bara sparas i det nuvarande scopet. Om man vill föra värden till det tidigare scopet, gör man det via ”Return” satser.

Både funktioner och variabler skapas och sparas i scope.

Kontrollsatser

Kombinationen av boolesk logik och jämförelser tillåter oss att skapa kontrollsatser. En av dessa satser är if-else-satsen. Denna sats utför en gren av kod om ett uttryck utvärderas till True, och en annan valfri gren om uttrycket utvärderas till False. Exempel nedan:

```
if a > b {
   c = 10          # Om a är större än b så tilldelas c värdet 10.
}
else {
   c = 20          # Annars så tilldelas c värdet 20.
}
```

Else-grenen ovan kan helt utelämnas om man vill.

En annan kontrollsats i språket är loop-satsen. Loop-satsens uppgift är att upprepa ett block med kod ett visst antal gånger. Denna implementation är en slags hybrid mellan klassiska for- och while-loopar. Exempel på användning nedan:

```
a = 5;
loop 5 {
   a++             # Höj värdet på a med 1, fem gånger om.
};

loop a < 50 {
   a++             # Höj värdet på a med 1, så länge a är mindre än 50.
}
```

I exemplet ovan visas loop-konstruktionen. Om man matar in ett heltal efter ”loop”, som i första exemplet, så upprepas koden inuti loopen så många gånger som heltalet. Detta är ungefär ekvivalent med en for-loop i andra programspråk.

I andra exemplet så matas ett booleskts (sant/falskt) uttryck in. Koden inuti loopen upprepas ända tills det värdet utvärderas till False.

Funktioner / Metoder

Automatot stödjer funktioner, även kallat metoder. Funktioner gör så att man kan strukturera kod på ett snyggare sätt, och används även för att undvika att man upprepar massor kod. Här är ett enkelt exempel:

```
# Här namnges funktionen till namnet min_funktion
fun min_funktion()
{
   # Denna kod utförs när man anropar funktionen
   Print(”Hello World”)
};


# Vi anropar funktionen vi skapade. Detta skriver ut ”Hello World”.
min_funktion();

# Vi skapar en till funktion, denna gången med ett parametervärde.
# Parametrar tillåter värden att matas in i funktionen.
fun min_andra_funktion(parameter)
{
   # Print tar nu emot parametern, så vi skriver ut det vi skickar in i funktionen!
   Print(parameter)
};

# Vi anropar den andra funktionen med parametern ”Hejsan svejsan”.
# Detta kommer skriva ut ”Hejsan svejsan”.
min_andra_funktion(”Hejsan svejsan”)
```

Funktioner kan alltså användas för att strukturera upp kod, och de kan även ta in parametervärden som man har tillgång till inuti funktionen.

En annan viktig del av funktionerna som vi stödjer är Return. Detta gör så att funktioner kan skicka tillbaka (returnera) värden! Nedan så skapas en funktion som tar in två stycken värden, a och b, adderar dem, och returnerar dem.

```
# VI skapar en funktion som tar in två parametrar.
fun addition(a, b)
{
   # Vi skickar tillbaka summan av de två parametrarna.
   Return a+b
};

# Vi anropar funktionen, och lagrar resultatet i variabeln c. c får värdet 12.
c = addition(5, 7);
```

Funktioner stödjer även standardvärden av parametrar, som appliceras om ingenting annat matas in. För att använda standardvärdet så ska Nil matas in på den platsen. Detta eftersom funktioner måste anropas med exakt det antal parametrar den kan ta emot. Exempel:

```
# Vi skapar en funktion med en parameter, som har standardvärdet 0.
fun add_one(a = 0)
{
   Return a+1
};

b = add_one(Nil);          # Vi anropar med Nil för att få standardvärdet, b blir tilldelad 1.
c = add_one(5)             # Vi skriver över standardvärdet, c blir tilldelad 6.
```

Standardvärdena behöver inte nödvändigtvis vara utryck, exempelvis 1, ”Test”, False, utan kan vara variabler eller funktionskall.

Följande är möjligt:

```
var = 2

fun foo(a=var) { Return a}

fun bar(b=foo())
```

Att notera är dock att funktioner som används som standardvärden får bara returnera en sak.

Automation och Inbyggda Funktioner

Syftet med språket var att automatisera små uppgifter på datorn, och detta är funktionerna som gör syftet möjligt. Det finns funktioner som berör muspekarens position:

```
# Flyttar musen till pixeln på skärmen med koordinat 200, 300. Övre vänstra hörnet är position 0, 0.
Move(200, 300);

# Lagrar muspekarens position i variablerna x och y.
x = MousePos(”x”);
y = MousePos(”y”);

# Ökar muspekarens x-koordinat med 200, och behåller samma y-koordinat.
Move(MousePos(”x”) + 200, MousePos(”y”));
```

Det finns funktioner som kan klicka på musens knappar:

```
Click();                   # Vänsterklickar en gång.
RightClick();              # Högerklickar en gång.
MiddleClick();             # Klickar med mushjulet en gång.
DoubleClick();             # Vänsterklickar två gånger.

MouseDown(”left”);         # Håller ner vänster musknapp.
MouseDown(”right”);        # Håller ner höger musknapp.
MouseDown(”middle”);       # Håller ner mushjulet.

MouseUp(”left”);           # Släpper vänster musknapp.
MouseUp(”right”);          # Släpper höger musknapp.
MouseUp(”middle”);         # Släpper mushjulet.
```

Det finns funktioner som klickar på tangentbordets tangenter:

```
# Send skriver ut text exakt så som det har matats in.
Send(”Hejsan”);              # Skriver ”Hejsan”, utan citationstecken.
a = ”God dag”;
Send(a);                     # Skriver ”God dag”, eftersom det var lagrat i variabel a.

# För mer minutiös kontroll, använd dessa funktioner istället för Send.
KeyDown(”shift”);            # Håller nere shift-tangenten.
KeyStroke(”k”);              # Klickar på k-tangenten.
KeyUp(”shift”);              # Släpper på shift-tangenten.
```

Dessa funktioner på egen hand utförs omedelbart, utan någon tanke på timing. Saker man försöker automatisera kanske då inte hinner med alla snabba kommandon som utförs, på grund av internethastighet, datorhastighet, med mera. För att lösa detta problem så inkluderas det en funktion som hjälper att halta på programspråket och vänta en tid. Detta är Sleep-funktionen, exempel nedan:

```
Sleep(500);                  # Sover i 500 ms, en halv sekund.

a = 2000;
Sleep(a);                    # Sover i 2000 ms, eftersom det var värdet på a.
Sleep(500 + 3*300)           # Sover i 1400 ms, eftersom det var resultatet på uttrycket.
```

Kommentarer

Språket stödjer kommentarer. Dessa kan fritt skrivas i källkod utan att något händer, de ignoreras helt enkelt. Detta är bra för dokumentation och anteckningar. Exempel:

```
a = 3          # Denna kommentar har ingen påverkan! a blir ändå tilldelad 3.

# b blir inte tilldelad 2 eftersom det är skrivet inuti en kommentar.
# b = 2

# Kommentarer gäller från #-tecknet till slutet av raden.
```

Allmänna kodregler

De flesta saker som man skriver i språket är så kallade uttryck. Dessa kan vara tilldelningar, automationsfunktioner, if-else-uttryck, loopar, med mera. Ett ”block” är en följd av uttryck. Kropparna av if-else, loop, och funktions-satser är alla block. En fil av källkod är också ett block. Om flera uttryck kommer efter varandra i ett block, måste de skiljas med semikolon (;). Exempel:

```
Uttryck
Uttryck;
Uttryck; Uttryck;
Uttryck; Uttryck;

fun foo(){Uttryck; Uttryck} ;

Uttryck
```

Kolla gärna tillbaka i användarmanualen på exempel hur semikolon används.

Sammanställning

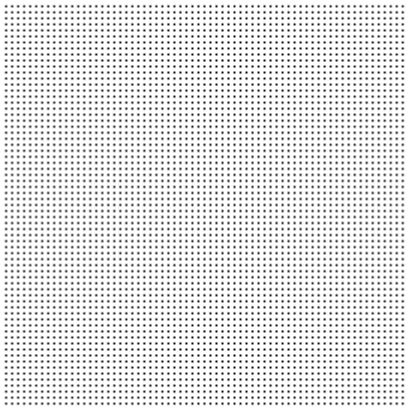
Med dessa funktioner så kan man göra nästan allting som du själv kan göra med en mus och ett tangentbord, det gäller bara att koda det!

Här är ett exempel och körresultat av lite kod som ritar en fin bild i Paint. Bilden var ritad med pennverktyget med minsta tjockleken. Resulterande bilden är till höger.

```
x = 200;
y = 200;

# Fokusera på Paint genom att klicka på det.
Move(x, y);
Click();

# Rita en fin bild.
loop x < 400
{
   y = 200;
   loop y < 400
   {
      Move(x, y);
      Click();
      y+=3;
      Sleep(1)
   };
   x+=3
}
```



Systemdokumentation

Översiktlig Beskrivning

Språket Automato använder sig av parsern rdparse.rb, denna och implementationen av språket är skrivet i programspråket Ruby.

Automatos syntax är inspererat av Ruby, Python, och C++. Implementationen/beteendet liknar mer Ruby och Python. Automato har även några unika beteenden, bestämda av skaparna, för att försöka underlätta användningen av språket.

Automatos kodbibliotek består av 7 huvudsakliga filer:

- Parsern (rdparse.rb)
- Syntaxreglerna (automato.rb)
- 3 stycken implementationsfiler (Assignment.rb, classes.rb, Func_call.rb)
- Testfil för att se så att de lovade funktionerna fungerar (tests.rb)
- Interface filen för Windows (auto_click.rb, även undermappen auto_click)

Grammatikens form

Grammatiken är beskriven i BNF (Backus-Naur-form). Formen förklaras snabbt nedan:

BNF-syntax:
::= Definieras som
| Eller
<> Kategorinamn
"" Sträng ska vara i matchningen t.ex "{" eller "}"
/<tecken>/ Reguljärt uttryck

Grammatiken

```
<parameter> ::= <identifier>

<func_name> ::= <identifier>

<print> ::= "Print" "(" <string> ")"
          | "Print" "(" <identifier> ")"

<send> ::= "Send" "(" <string> ")"
          | "Send" "(" <identifier> ")"

<mouse_move> ::= "Move" "(" <a_expr> "," <a_expr> ")"

<mouse_click> ::= "Click" "(" ")"
                | "RightClick" "(" ")"
                | "MiddleClick" "(" ")"
                | "DoubleClick" "(" ")"

<mouse_down> ::= "MouseDown" "(" <string> ")"
```

```
                    | "MouseDown" "(" <identifier> ")"

<mouse_up> ::= "MouseUp" "(" <string> ")"
                    | "MouseUp" "(" <identifier> ")"

<key_down> ::= "KeyDown" "(" <string> ")"
                    | "KeyDown" "(" <identifier> ")"

<key_up> ::= "KeyUp" "(" <string> ")"
                | "KeyUp" "(" <identifier> ")"

<key_stroke> ::= "KeyStroke" "(" <string> ")"
                    | "KeyStroke" "(" <identifier> ")"

<if_stmt> ::= "if" <expression> "{" <stmt_list> "}"  "else " "{" <stmt_list> "}"
                | "if" <expression> "{" <stmt_list> "}"

<loop_stmt> ::= "loop" <expression> "{" <stmt_list> "}"

<sleep_stmt> ::= "Sleep" ( <expression> )

<return_stmt> ::= "Return"
            | "Return" <expression_list>

<augmented_assignment_stmt> ::= <target_list> <aug_op> <expression_list>

<aug_op> ::= "-="
                | "+="
                | "*="
                | "/="
                | "&="
                | "//="
                | "%="

<assignment_stmt> ::= <target_list> "=" <expression_list>

<target_list> ::= <target_list> "," <target>
                    | <target>

<target> ::= <identifier>
```

```
<expression_stmt> ::= <expression_list>

<expression_list> ::= <expression>
                    | <expression_list> "," <expression>

<expression> ::= <conditional_expression>

<conditional_expression> ::= <or_test>
                           | <or_test> "if" <or_test> "else" <expression>

<or_test> ::= <and_test>
            | <or_test> "or" <and_test>

<and_test> ::= <not_test>
             | <and_test> "and" <not_test>

<not_test> ::= <comparison>
             | "not" <not_test>

<comparison> ::= <a_expr>
               | <a_expr> <comp_operator> <a_expr>

<comp_operator> ::= "<"
                  | ">"
                  | "<="
                  | ">="
                  | "=="
                  | "!="

<a_expr> ::= <math_term>
           | <a_expr> "-" <m_expr>
           | <a_expr> "+" <m_expr>

<m_expr> ::= <power>
           | <m_expr> "/" <power>
           | <m_expr> "*" <power>
           | <m_expr> "%" <power>
           | <m_expr> "//" <power>
```

```
<power> ::= <self_mod> "^" <power>
          | <self_mod>


<self_mod> ::= <atom> <self_mod_op>
             | <atom>

<self_mod_op> ::= "++"
                | "--"
                | "**"
                | "^^"

<primary> ::= <func_call>
            | <atom>

<func_call> ::= <primary> "(" <argument_list> ")"
              | <primary> "(" ")"

<argument_list> ::= <positional_arguments> "," <argument_list>
                  | <positional_arguments>

<positional_arguments> ::= <expression>

<atom> ::= <mouse_pos>
         | <literal>
         | <enclosure>
         | <variable>

<variable> ::= <identifier>

<identifier> ::= /[a-zA-Z_]+/

<enclosure> ::= "(" <expression_list> ")"

<literal> ::= <bool>
            | <string>
            | <float>
            | <integer>

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

<string> ::= /"(?:(?!").)*?"/

<float> ::= <integer> "." <integer>
| "-" <float>

<integer> ::= Integer
| "-" Integer

Grammatikens delar

Lexikalisk analys:
Parsern rdparse.rb kommer med en tokenizer inbyggt, vilket är den lexikaliska funktion Automato använder.
Inuti rdparse finns en "token()" funktion. Den använder sig av reguljära utryck för att strukturera en given sträng till tokens som parsern sedan använder.

Parsing:
Parsern använder sig utav den givna sekvensen av tokens för att matcha enligt alla Automatos regler. Av dessa matchningar skapas ett Abstrakt syntax/klassträd. Nästan varje matchning skapar en egen nod i detta abstrakta syntax/klassträd. Vissa regler är dock bara i språket för att förtydliga strukturen.
Match-grammatiken är skriven med BNF struktur, och är dokumenterat ovanför.

Evaluering:
När hela token sekvensen har matchat korrekt, och det abstrakta syntaxträdet är klart, evalueras det. I evaluering evalueras det givna syntaxträdet. Här beräknas, sparas och skapas de beskrivna uttrycken, variabeldeklarationerna, funktionsdefinitionerna, med mera. Noden högst upp i syntaxträdet evalueras först, och den i evaluerar i sin tur de noder som ligger under denna nod. Detta skapar en dominoeffekt mellan noderna, och programmet evalueras som man förväntar sig.

Installation och användning

Programspråket är just nu inte tillgänglig för nedladdning, detta kommer säkert ändras i framtiden. Håll utkik på TDP019's kurshemsida så finns det säkert en nedladdning där någon dag.

Innan språket kan användas så måste språket som Automato vara utvecklad i vara installerad. Installera Ruby från https://www.ruby-lang.org/en/downloads/. Notera att du måste använda ett Windows-baserat operativsystem för att språket ska fungera som det ska.

När du har Ruby installerat och språkets filer nedladdade så är det dags att koda! Skriv din kod i en textfil och spara den på ditt system. För att exekvera koden så öppnar du en terminal / kommandotolk, navigerar till mappen där automato.rb ligger, och sedan skriver ”ruby automato.rb <CODE.TXT>” där <CODE.TXT> är sökvägen till din kodfil.

Erfarenheter och reflektion

Parsern vi använde är samma som gavs ut till vissa uppgifter i TDP007, rdparse.rb. Den var väldigt smidig att använda. Parsern tog emot nästan ren BNF-grammatik och betedde sig så man förväntade sig. Loggfunktionerna var även extremt hjälpsamma om matchningar inte fungerade så de skulle.

Det första tillvägagångssättet vi testade stötte på problem senare i projektet. Det sättet parsade koden, och utförde programlogik precis när viss grammatik matchade.

Vi byggde alltså inget abstrakt syntax/klassträd utanför parsen, utan utförde logik precis där matchningarna hände. Detta stötte på problem när flera delmatchningar passade, vissa instruktioner kunde utföras flera gånger, och det var extremt knöligt att få det pålitligt.

För att lösa detta problem så byggde vi om match-logiken en del. Nu så byggs ett slags klassträd upp med föräldrar och barn-noder. Alla dessa klasser har en evaluate-funktion som gör vissa grejor.

När en hel kodfil har parsats så får vi en toppnivå-klass i trädet. Exekverar vi evaluate på den så går en domino-effekt genom hela trädstrukturen och programmet utför instruktioner så som man skulle förvänta sig.

Denna klassombyggnad var väldigt enkel att utföra. Vi hade en rätt så väldefinierad BNF-grammatik till språket innan vi påbörjade utvecklingen, så det var egentligen bara att ändra vad som hände när grammatiken matchade.

Att vi hade en så pass väldefinierad BNF har hjälp oss mycket igenom hela processen, då vi till exempel kunde utnyttja oss av att se vilka saker som berodde på varandra, och då se i vilken ordning sakerna skulle implementeras. Samt kunde vi lätt se vilka saker som eventuellt kunde avvecklas.

Vi lärde oss även en del om hur Ruby's gem-funktioner fungerar. Innan så kändes det lite magiskt, man bara skrev 'gem install auto_click' i terminalen, och så fungerade resten automatiskt.

Gemet som vi använde oss av för automation heter auto_click. Det gemet var lite föråldrat och hade inte fullt stöd i den moderna Ruby-versionen vi använde, så vi var tvungna att göra en liten fix i dess källkod för att det skulle fungera väl. Vi lärde oss då var Ruby-gems är lagrade, och insåg även att det inte är magi, det är helt enkelt Ruby-kod med lite gem-filer på toppen! Efter fixen så flyttade vi källkoden för gemet in i vår projektmapp, och fick det att fungera utan att en 'gem install' behövdes.

Under projektets gång så har vi insett fördelarna med att föra enhetstester. Redan tidigt i projektet så var det enkelt att ta sönder gamla funktioner när man utvecklade nya funktioner. Grammatik kunde enkelt gå sönder, returvärden kunde bete sig på oväntade sett, med mera. Vi började då föra en test-fil som innehöll omfattande tester för alla funktioner i språket. I den filen matade vi in kod, och vad för värden vi förväntade oss från den kodexekveringen. Innan vi var nöjda med nya funktioner så körde vi testfilen för att ta reda på om saker hade gått sönder.

Vi har dock även insett att man inte alltid ska vara lat när man skriver testerna. Då till exempel denna kod: "a=1; a,b += 1,2" fungerade.
Detta berodde på att ett test innan har skapat variabeln b. Att detta var fallet gömde felet att denna kod inte borde fungera, vilket tog oss två veckor att inse.

Språkspecifikationen har en del skillnader mot den slutgiltiga produkten.

En av dessa skillnader var att lyssna efter tangentbordsknappar och utföra kod när dessa upptäcks. Vi kunde inte hitta något sätt att effektivt lyssna efter tangentbordstryck, och även om vi gjorde det så hade vi ingen aning om hur man skulle "blockera" det knapptrycket och utföra andra funktioner istället. Slutgiltiga språket måste exekveras genom en terminal.

Scope i språket liknar det som i Python/Ruby. Skillnaden vi har implementerat är att variabler/funktioner kan nås från tidigare scope. En funktion skulle kunna använda sig av variabler från det ställe den kallades ifrån. Dock så liknar vi Python/Ruby i det att variabler skapas bara i det lokala scopet, och kan bara flyttas mellan scope med hjälp av return. Att funktioner har lokalt scope så gjorde det effektivt pass-by-value, i slutändan.

I exempelkoden i språkspecifikationen så hade vi radbrytning (\n) som skiljare mellan uttryck, eller semikolon (;) som skiljare om man skriver många uttryck på samma rad. Detta fick problem i slutändan då vi i vissa fall ville att radbrytning skulle ha betydelse (som skiljare), och i vissa fall inte ha någon betydelse alls (som när man skriver med block.). Med radbrytning som skiljare så var man tvungen att ha alla kontrollsatser som enradare, vilket inte är rimligt. I slutändan så blev semikolon skiljaren mellan uttryck, whitespace spelar ingen roll alls. Om det står flera uttryck i rad så behöver man semikolon på alla uttryck förutom det sista, om det står ett enda uttryck i ett block, behövs inget semikolon alls. Exempel:

```
a = 2;
if a < 3 { c = 5 };
d = 6
```

Automationsfunktionerna är rätt likt det som står i specifikationen, men med några skillnader. Send()-funktionen tar inte emot specialtangenter. Den tar istället emot en sträng som skrivs ut exakt som den skickades in, med stora bokstäver och specialtecken automatiskt formaterade. Vi lade till KeyStroke(), KeyDown(), och KeyUp()-funktioner som kan användas för mer minutiös kontroll. De kan ta emot enskilda tangenter och hålla ner/släppa de som man vill. Exempel:

```
KeyDown('shift');
KeyStroke('k');
KeyUp('shift')
```

Koden ovan skriver ut versalen K.