Nils Karlsson och Jesper Larsson

Linköpings universitet
Linköping
6 maj, 2010

Innehållsförteckning

1 Inledning

Denna rapport är skriven som en del av examinationen till kursen TDP019, en projektkurs om datorspråk. Projektgruppen består av två studenter på IP-programmet, Nils Karlsson och Jesper Larsson.

I stora drag består rapporten av 3 delar:

- **Användarhandledning**. Här beskriver vi språket, vilka konstruktioner som är möjliga, hur syntax ser ut samt ger exempel på hur man använder dessa.
- **Systemdokumentation**. Denna del är tekniskt orienterad. Om användarhandledningen förklarar vad språket är förklarar denna del varför det är på just det sättet. Denna del innehåller även översikt över hur koden är strukturerad.
- **Utvärdering**. Bägge författarna har skrivit var sin reflektion där vi utvärderar projektet, vad vi tyckte och eventuellt vad vi kan tänka på att göra bättre till nästa projekt. Denna del innehåller även en rad synpunkter (ca 20 stycken totalt i de bägge reflektionerna) om konstruktion av programspråk.

En viktig del av vårt projekt har varit beslutet att implementera en egen parser och lexer från grunden istället för att använda färdiga verktyg. Vi anser att vi lär oss avsevärt mycket mer på detta vis, även om det är avsevärt mycket mer tidskrävande. Vårt mål med detta projekt har varit att utveckla ett elegant språk som går enkelt att skriva och som samtidigt har en syntax som är både tydligt och lättläst.

Eftersom språket är baserat runt filosofin att symboler används snarare än nyckelord har vi döpt vårt språk till ”Sym”, vår lexer till SymLexer och vår parser till SymParser.

Denna dokumentation täcker version 1.0 av Sym.

2 Användarhandledning

Denna sektion innehåller kortare beskrivningar av konstruktionerna i Sym samt en förklaring till hur de bör användas.

2.1 Introduktion

Denna användarhandledning är främst menad som en introduktion till själva språket, inte till programmering. Vi förklarar därför inte de begrepp som används då vår målgrupp främst innefattar folk med tidigare programmeringskunskap.

Ett genomgående tema i Sym är / \-notationen för att markera scope, detta är något man bör vänja sig vid och känna igen. Det är vad som står ihop med dem som bestämmer vilken konstruktion som avses(/? Villkor, /# kommentar, /$ funktion m.m.). Detta kommer framgå tydligare när vi introducerar konstruktioner så som funktioner, klasser, villkor m.m.

2.2 Krav

För att kunna använda Sym krävs en Ruby-interpretator av version 1.9 eller högre. Detta kan laddas ner ifrån:
www.ruby-lang.org
Instruktioner hur Ruby installeras finns på denna hemsida.

Vidare kräver Sym en texteditor som klarar av att spara ner i vanligt textformat utan formatering (dvs. inte OpenOffice eller Word), Notepad eller en IDE (eclipse/netbeans/emacs) fungerar väl. Vi rekommenderar dock editorn jEdit, för vilken vi utvecklat ett plugin som tillhandahåller syntaxhighlighting. Instruktioner hur detta installeras gås igenom i ett senare avsnitt av användarhandledningen.

2.3 Ditt första Symprogram

Det bästa sättet att lära sig ett nytt programmeringsspråk är att dyka rakt in med ett exempel, det mest klassiska exemplet är ”hello world”-exemplet, vilket gör att det passar bra.
Att skriva ut i Sym är mycket enkelt. Öppna valfri texteditor och skriv in följande:

```
<< ”Hello world”
```

Alternativt kan man använda printoperatorn som inte gör newline efter sig:

```
<<< ”Hello ”
<<< ”world”
```

Spara filen som helloworld.sym, gärna i samma mapp som Sym ligger. Gå sedan till mappen via en kommandotolk/terminal och skriv:

ruby sym.rb helloworld.sym

OBS att detta kräver att ruby ligger i din PATH-systemvariabel, om installationen av Ruby gick som den ska gjordes detta automatiskt.

2.4 Kommentarer

Det finns två typer av kommentarer i Sym: enradskommentarer och flerradskommentarer.

```
# Jag är en enradskommentar

#/
     Jag är en flerrads-
     kommentar
\#
```

Kommentarerna kommer att ignoreras av Sym. För närvarande finns det inget verktyg tillgängligt för att extrahera kommentarer från koden och generera ett separat dokument med kommentarer i likhet med JavaDoc. Detta är dock planerat till nästa version av Sym.

2.5 Operationer

2.5.1 Uttrycksoperatorer

Här nedanför sammanställs de operatorer som finns tillgängliga i Sym:

(*x och y denoterar två värden; resten är operatorer*)

Operator	Beskrivning
x || y	Logiskt OR (disjunktion)
x && y	Logiskt AND (konjunktion)
! x	Logisk NOT (negation)
x < y, x <= y, x > y, x >= y, x == y, x != y, x === y	Jämförelseoperationer; evalueras till sanningsvärden
x – y, x + y	Subtraktion, Addition
x * y, x / y, x % y	Multiplikation, Division, Modulo
x ** y, x ^ y	Power (upphöjd till; x^y)

2.5.2 Operatorprecedens

Operatorprecedens innebär vilka operatorer som har företräde framför andra, dvs. vilka som evalueras först. Detta kallas även operatorprioritet. Operatorprioriteten är för närvarande ofullständig i Sym, vilket innebär att i vissa operationer finns operatorprioritet, medan i andra uträkningar saknas detta.

```
x = 1 + 2 * 3        # => 7
x = (1 + 2) * 3      # => 9
x = 2 * 3 ** 4       # => 1296. Bör egentligen vara 182.
```

Man bör alltså inte förlita sig på operatorprioriteten, utan placera parenteser för att förtydliga prioriteten.

2.5.3 Numeriska operander

Ovanstående operatorer är samtliga applicerbara på numeriska operander.

2.5.4 Logiska operationer

Logiska operatorer går att använda på alla primära typer (strängar, flyttal, heltal, m.m.). Alla värden som är ekvivalenta med noll (det numeriska talet 0) returnerar falskt, medan övriga värden (som inte är booleska) returnerar sant. De logiska operationerna följer annars samma regler som för matematiken.

```
<< 1 == 1                 # => true
<< (1 == 1) || (2 == 1)   # => true
<< (1 == 1) && (2 == 1)   # => false
```

2.5.5 Operationer på strängar

Det är möjligt att använda jämförelseoperationerna == och != på två strängar. Övriga jämförelseoperationer är förbjudna. Logiska operationer på strängar är tillåtet. Övriga operationer är förbjudna, förutom x * y som returnerar en multiplicerad konkatenering av strängen, ifall y är ett heltal:

```
<< "Hej" * 3   # Ger: HejHejHej
<< 3 * "Hej"   # Förbjudet: Ger felmeddelande.
```

2.5.6 Modifierande operatorer

En modifierande operator är egentligen en förkortning för vanligt förekommande operationer. Här sammanställs de modifierande operatorer som finns i Sym:

(*x och y denoterar två värden; resten är operatorer*)

x += y	Addition. Motsvarar x = x + y
x -= y	Subtraktion. Motsvarar x = x – y
x *= y	Multiplikation. Motsvarar = x * y
x /= y	Division. Motsvarar x = x / y
x %= y	Modulus. Motsvarar x = x % y
x **= y	Power. Motsvarar x = x ** y
x ^= y	Synonym för Power. Se ovanför.

Dessa operationer följer samma regler som för uttrycksoperatorer. Dessa är förkortningar för vanligt förekommande numeriska operationer.

Precis som för strängar gäller det att x*= y returnerar en multiplicerad sträng. Se exemplet nedanför.

```
x = "Hej"
x *= 3          # Ekvivalent med x = x * 3
<< x            # => HejHejHej
```

2.6 Villkorssatser

Villkorssatser i Sym liknar de i andra språk – det är ett sätt att välja bland olika satser innanför ett if-block, beroende på om de associerade villkoren uppfylls eller ej.

2.6.1 Generella formatet

Det generella formatet för if-satser ser ut så här:

```
?/ <villkor>            # inleder if-block
     <satser>
\?/ <villkor>           # else-if (valfri)
     <satser>
\*                      # else (valfri)
     <satser>
\?                      # avsluta if-block
```

För dem som inte är bekanta med if-satser kan detta inte säga särskilt mycket. För den som vill veta mer om hur if-satser ska användas, läs vidare i nästa sektion där vi går igenom if-satserna utförligt.

2.6.2 If-satser i praktiken

If-satser är alltså en grundläggande konstruktion i alla programmeringsspråk. Hur de kan användas i Sym demonstreras enklast med ett exempel:

```
<< "Skriv in värdet av a:"
>> a
?/ a == 0
     << "Värdet av a är 0!"
\?/ a == 1
     << "Värdet av a är 1!"
\?/ a == 2
     << "Värdet av a är 2!"
\*
     << "Värdet av a är inte 0 eller 1!"
\?
```

Vi läser in indata från användaren och sparar det i variabeln a. Därefter går vi in i en if-sats, där vi först jämför om a är ekvivalent med noll. Om den är det skriver vi ut "Värdet av a är 0!".

Därefter kommer de så kallade else-if-villkoren. Om det inledande villkoret (a == 0) är falskt, så körs else-if villkoren i tur och ordning. Om något av de villkoren uppfylls så körs den koden som kommer därefter och if-satsen avbryts. Till sist kommer else-villkoret, som körs ifall de andra villkoren i hela if-satsen inte uppfylls. Detta är alltså "den sista utvägen".

Notera att vid definitionen av en hel if-sats är den första if-satsen obligatorisk medan else if- och else-satserna är valfria.

2.7 Funktioner

Funktioner i Sym fungerar något annorlunda än många andra programmeringsspråk. I Sym tillåts du som användare använda både statisk och dynamisk typning. Med andra ord kan du tvinga att vissa delar av funktionen är av en viss typ, medan vissa delar lämnar du fritt (som ett dynamiskt typat språk).

Funktioner i Sym följer följande format:

```
$/returtyp namn(datatyp argument)
     funktionskod
\$
```

Både returtypen och datatypen för argument är frivilliga att ange. Om de inte anges antas de vara Void, det vill säga ingen typkontroll utförs. Då kan man alltså skicka vad man vill.

Nedan följer ett komplett exempel på hur en funktion kan se ut:

```
$/Float fun(Int int_arg, void_arg, String str_arg)
     << int_arg
     << void_arg
     << str_arg
     return 1.0
\$
<< fun(5, "vadsomhelst", "str")
```

Notera att det andra argumentet inte har någon datatyp specificerad. Detta innebär att det andra argumentet kan vara vad som helst, ingen kontroll utförs från språkets sida. De andra två argumenten, och returvärdet, tvingas dock vara sina respektive datatyper.

Självklart kan de kombineras i all oändlighet. Man kan strunta i all data, bara strunta i ett argument, strunta i returvärdet, inte ha några argument alls och så vidare.

Sym har även stöd för nästlade funktioner, det vill säga lokala funktioner som endast existerar i varandra. Ett exempel på detta är:

```
$/Int outer_fun(Int arg1)
     $/Int local_fun(Int local_arg)
          return local_arg+1
     \$
     return local_fun(arg1)*10
\$
<< outer_fun(5)
```

Kör man denna kod kommer den evalueras som (5+1)*10 = 60. Funktionen *local_fun* kommer inte

vara tillgänglig utanför *outer_fun*-funktionen.

2.8 Objektorientering

Klasser i Sym är ganska likt de i flesta andra språk. Men det kan ändå finnas vissa saker man kan behöva vänja sig vid som ny till Sym. I Sym är inte objektet uppdelat i en privat och en publik del. Istället är alla instansvariabler privata och alla metoder är publika.

Detta är ett designval vi gjort mest för att dra ner på utvecklingstiden. Sym har inte heller stöd för arv eller att utöka andra klasser. Detta betyder dock inte att de inte går att använda. Vi anser att det går göra det mesta man är van vid även i Sym även om vissa delar blir jobbigare.

En klass definieras enligt följande mall:

```
@/Klassnamn
     lista med instansvariabler
     $/init()            # Jag är en konstruktor
     \$
\@
```

Som kommentaren antyder är init-metoden konstruktorn, det vill säga den metod som anropas för att skapa objektet. Denna metod måste alltid vara definierad, även om den är tom.

Man skapar sedan ett objekt, en instans av klassen, genom följande:

```
x = Klassnamn.init(argument)
```

Anropet till init fungerar som ett vanligt funktionsanrop, man kan alltså ange eventuella argument där om man föredrar det. När väl objektet är skapat kan man anropa alla metoder som fanns i den klass den skapades från på samma sätt som i de flesta språk:

```
x.metod(argument)
```

För den som har svårt att följa pseudokoden ovan kommer här ett riktigt, fungerande, kodexempel:

```
@/Person
     msg       # Instansvariabler
     id_nr

     $/init(String arg_msg, Int arg_id)
          msg = arg_msg
```

```
            id_nr = arg_id
        \$
        $/String get_msg()
            return msg
        \$
\@
p = Person.init("Sym", 42)
<< p.get_msg()
```

Här skapar vi alltså en klass Person som håller två instansvariabler (msg och id_nr). I denna klass skapar vi även en metod, *get_msg()*, som hämtar msg-variabeln och returnerar den som en sträng.

Under klassdeklarationen skapar vi ett objekt av Person, där vi skickar in två argument till konstruktorn. En sträng, ”Sym”, och ett heltal, 42 som konstruktorn sparar i detta objekt. Sist gör vi ett anrop till get_msg() metoden på objektet vi nyss skapade, och skriver ut resultatet av denna. Som förväntat skriver den ut strängen ”Sym”, samma som skickades in till konstruktorn.

2.9 Repetitionssatser

För att repetera en mängd av satser behövs en s.k. slinga. Vi kommer numera kalla det för en *loop*. En loop är en konstruktion som repeterar satser utifrån villkor. Det finns två typer av loopar implementerade i Sym, for-loopar och while-loopar.

2.9.1 Generella formatet

Det generella formatet för while-loopar ser ut så här:

```
|/ <villkor>
    <satser>
\|
```

Detta säger att <satser> skall upprepas så länge <villkor> uppfylls.

Det generella formatet för for-loopar ser ut så här:

```
../ min_värde max_värde var
    <satser>
\..
```

Detta säger att <satser> skall upprepas ett antal gånger, mer specifikt ((max_värde – min_värde) + 1) gånger och spara varje stegning i variabeln var.

Läs vidare i nästa två sektioner för hur dessa loopar ser ut i praktiken.

2.9.2 While-loopar

En while-loop exekverar ett block av satser iterativt så länge ett associerat villkor uppfylls. Detta villkor evalueras i början av loopen. Om villkoret är falskt till att börja med så kommer blocket av satser aldrig att exekveras. While-loopar demonstreras med följande exempel:

```
a = 0
|/ a != "q"
     << "Skriv in ditt meddelande (q för att avsluta)"
     >> a
     << "Du skrev in:"
     >> a
\|
```

Eftersom a är 0 till att börja med kommer programmet gå in i while-blocket där den frågar användaren efter ett meddelande. Denna fråga komma upprepas så länge svaret inte är "q".

En evighetsloop kan skrivas genom att ange ett villkor som alltid evalueras till sant. Ett exempel på ett sådant är "1 == 1". En evighetsloop ser ut så här:

```
|/ 1 == 1
     << "Oändlig loop!"
\|
```

2.9.3 For-loopar

for-loopar är mer lämpade (än while-loopar) för att upprepa ett block av satser ett bestämt antal gånger. När en for-loop initieras anger man hur många gånger den ska köras genom att sätta ett minimum-värde och ett maximum-värde. För varje stegning i loopen plussas detta värde på med 1 tills den når maximum-värdet. Detta värde sparas i en associerad variabel. Detta demonstreras bäst med ett exempel:

```
sum = 0
../ 1 10 i
     sum += i
\..
<< sum          # => 55
```

Vi definierar variabeln sum och tilldelar den värdet 0. Därefter skapar vi en for-loop som skall köras 10 gånger och för varje stegning från 1 till 10 spara värdet i variabeln *i*. Detta värde lägger vi till sum för att på så sätt räkna ut summan av 1 till 10. Utskriften blir alltså 55.

Notera att maximum-värdet måste vara högre än minimum-värdet eller vara ekvivalent. Annars skrivs ett fel ut och programmet avslutas. Om värdena är ekvivalenta exekveras det associerade blocket exakt en gång. Om vi t.ex. har:

```
../ 1 1 i
     << "Hello World"
\..
```

kommer detta skriva ut "Hello World" en gång. Detta enligt formeln ((max_värde – min_värde) + 1)

vilket ger (1 – 1) + 1 som är 1.

Till sist behöver vi notera att variabeln i endast har lokal räckvidd, dvs. att den endast är tillgänglig innanför det associerade for-blocket. Så fort for-blocket avslutas tappar variabeln i sitt värde och kan ej längre refereras. Det går alltså **inte** att skriva:

```
../ 1 10 i
     << "Hello Sym"
\..
<< i
```

2.9.4 Break-operatorn

Det är möjligt att hoppa ut ur en loop genom att använda den s.k. break-operatorn. Det går att använda den både för for- och while-loopar. Om den förekommer utanför en loop skrivs ett felmeddelande ut. Den hoppar bara ut ur den närmaste loopen. Break-operatorn kan användas så här:

```
sum = 0
|/ 1 == 1
     sum += 1
     ?/ sum > 10
          <--
     \?
\|
```

2.10 Specialkonstruktioner

Sym innehåller ett flertal speciella variabler, konstanter, metoder m.m. Se rubrik 3.7 för en genomgång av dessa.

2.11 Sym Mode

För närvarande går det endast att få stöd för highlighting och indentering av Sym-kod i programmeringseditorn jEdit. Här ska vi vissa hur man kan installera jEdit och en s.k. Sym mode som definierar hur koden skall highlightas och indenteras av editorn.

2.11.1 Installera jEdit

jEdit är utvecklat i Java och kan därmed köras på alla plattformar som har JRE (Java Runtime Environment) installerat. jEdit kan hämtas på den här länken: http://www.jedit.org/index.php?page=download

2.11.2 Installera Sym mode

För att installera Sym mode behöver du först mode-filen som går att finna i Sym-paketet (%SYM %/doc/modes/jedit/sym.xml, där %SYM% är sökvägen till Sym-katalogen). Denna fil ska du kopiera över till jEdit-katalogen, närmare bestämt %JEDIT%/modes, där %JEDIT% är sökvägen till var jEdit finns installerat. Klistra in sym.xml i denna katalog.

Därefter sök upp en fil i samma katalog vid namn *catalog*. Öppna denna fil. Lägg till följande rader:

```
<MODE NAME="sym" FILE="sym.xml"
     FILE_NAME_GLOB"*.sym" />
```

Detta medför att alla filer som har filsuffixet .sym kommer tolkas som Sym-filer. Spara catalog och stäng ner den. Öppna därefter jEdit. Gå in på **Utilities** > **Troubleshooting** > **Reload Edit Modes** (alternativt tryck in Alt+U+T+R i ordningsföljd). Skapa nu en fil med filsuffixet .sym, t.ex. test.sym. Klistra in följande kod:

```
../ 1 10 i
     << i
\..
```

Om du har gjort rätt ska de olika konstruktionerna markeras med olika färger. Om det inte fungerar, gå tillbaka till de olika stegen och se om du har missat något.

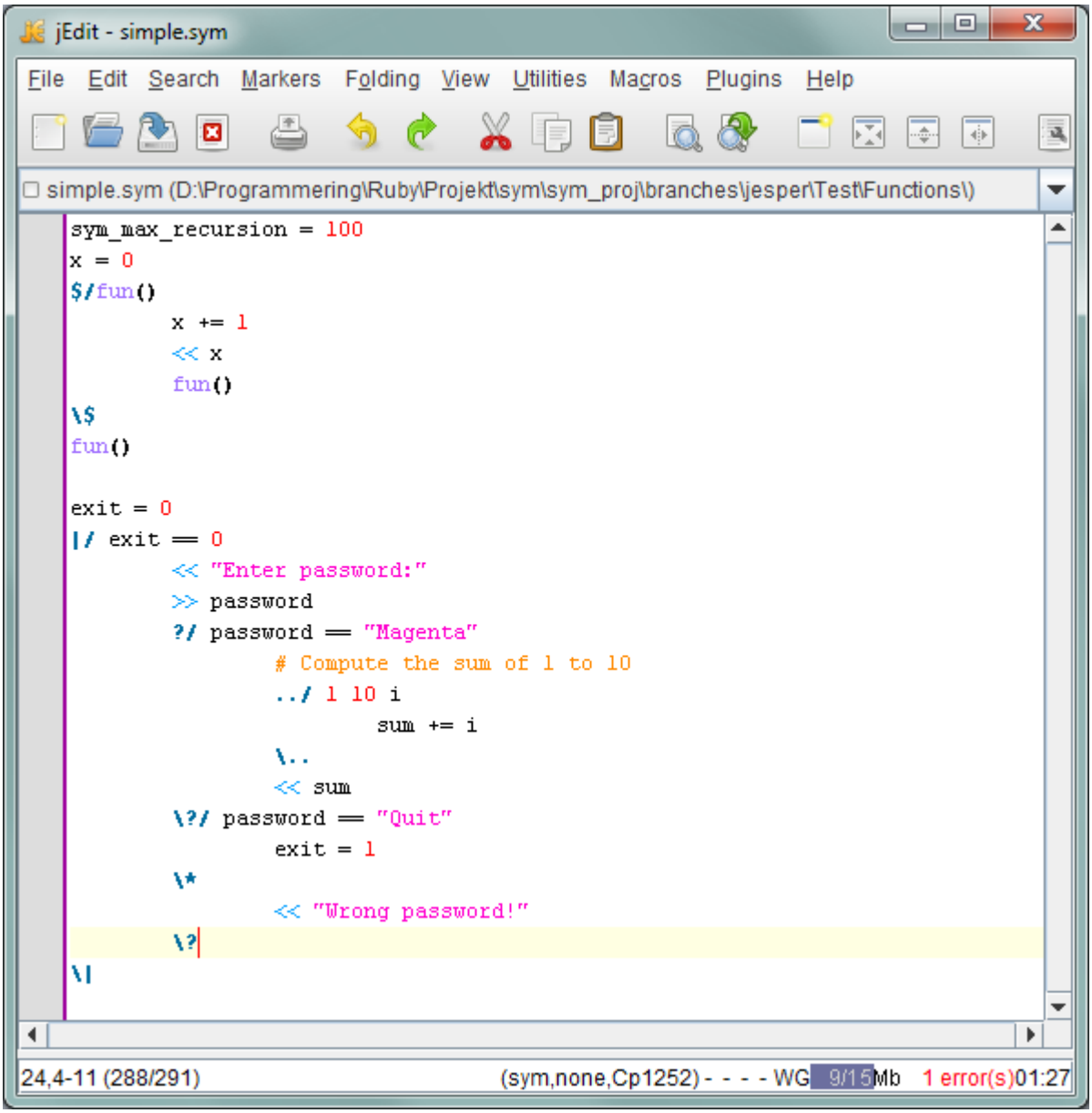

Illustration 1: Highlighting av Symkod i jEdit

3 Systemdokumentation

Systemdokumentationen är till stor del tekniskt orienterad. Här beskrivs hur implementationen fungerar, några av de designöverväganden som gjordes

3.1 Överblick

Implementationen av Sym är skriven i programspråket Ruby. Hela implementationen är konstruerad från grunden av oss själva, utan att använda färdiga verktyg. Detta har gjort att vi fått en ganska bra känsla för hur språk är uppbyggda som vi troligen inte fått om vi använt färdiga verktyg. Vi har även större kontroll över strukturen, hade vi använt rdparser eller liknande verktyg hade vi varit tvungna att hålla oss till de regler som är uppsatta av den.

Sym är uppbyggt av två delar. En större del, parsern, som är drivmotorn bakom allt. Den bestämmer vad som ska göras, den tvingar stor del av korrekt syntax och den kör även koden när den väl bestämt vad som ska göras.

En stor fördel av att ha skrivit implementationen i Ruby har varit att stor del av utvecklingen gått smidigt. Stödet i Ruby för reguljära uttryck är utmärkt, även om det var lite besvärligt att Ruby 1.9 eller över krävs för att kunna använda sätta namn på captures. Därmed blev vissa delar av koden inte riktigt lika innovativa som dem kunde ha varit, eftersom vi inte ville bryta kompabilitet med äldre versioner av Ruby.

Ett paket med samtliga filer tillhörande projektet kan laddas ner genom följande adress:
`http://www-und.ida.liu.se/~jesla813/tdp019.zip`

Implementationen är fördelad över följande filer:
sym.rb
Vår mainfil som läser in en fil med kod och skickar vidare denna kod till parsern. Fångar även eventuella fel och ger användaren möjlighet att köra om programmet i debugläge.

Backend/lexer.rb
lexer.rb innehåller koden för SymLexer.

Backend/parser.rb
parser.rb innehåller koden för SymParser.

Backend/datatypes.rb
Lista med olika datatyper vi använder för att identifiera olika delar. Exempelvis finns här klassen för ett ”lexem” som används för att identifiera särskilda konstruktioner och vår interna lagring av klasser, objekt m.m. De flesta av dessa är ganska triviala.

Backend/scope.rb
Innehåller klassen för scope, som parsern använder för att lagra data. Denna är en viktig del och fick därför en egen fil.

3.2 SymParser

SymParser är vår implementation av parsern till vårt språk. Den är drivmotorn för själva tolkningen av språket men kan inte själv köras, utan måste anropas från vår mainfil (sym.rb).
SymParser förlitar sig på SymLexer för att identifiera vad varje enskild bit är. Skulle vi vilja byta syntax för vårt språk skulle endast lexern behöva ändras och parsern förbli oförändrad.

Parsern har en huvudloop där den förväntar sig någon typ av ”statement”. Den anropar lexern och frågar vad nästa lexem är för något. Därefter kollar den sedan mot en lista vad som är ett giltigt statement. Beroende på vilken typ av statement det är körs sedan en funktion i parsern som är unikt för just detta statement. Denna funktion hämtar eventuellt sedan fler lexems beroende på vilken typ av statement det är, och så vidare. Detta pågår tills slutet av filen har uppnåtts. Nedan följer illustrativ pseudokod:

Oändlig loop:
    Hämta lexem, spara som lexem

    Om lexem är EOF (end of file):
        break
    Annars om lexem är printoperator:
        print_operator(lexem)
    Annars om lexem är classdef:
        class_def(lexem)
    ...
    Annars:
        Skriv ut ett felmeddelande, matchade inget av det som förväntades

Bild 3.2.1 visar ett enkelt anrop för hur nedanstående utskrift hanteras och hur parsern kommunicerar med lexern.
<< ”exempel”

Parsern kör hela tiden get_next_lexem() funktionen för att få nästkommande lexem.
Först får den ett PrintOperator lexem, vilket innebär att parsern kommer förvänta sig något den kan skriva ut härnäst. Den tar då mot en String, med värdet ”exempel”.
Till sist mottages ett EndOfFile lexem, så parsern avslutas.

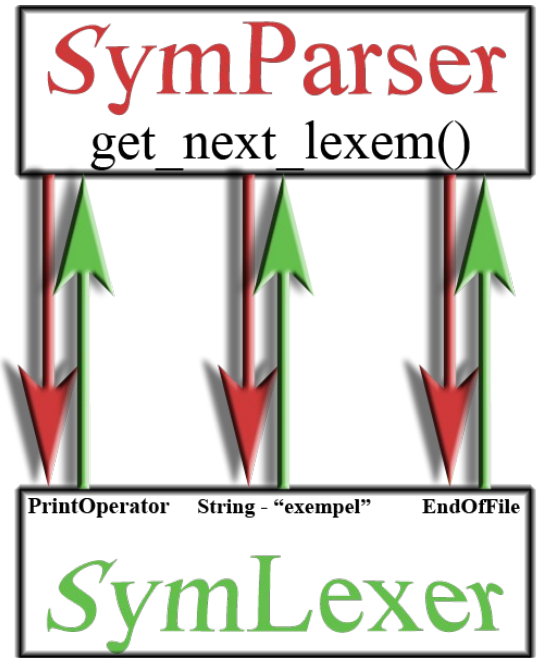

Bild 3.2.1 Kommunikation mellan SymParser och SymLexer

3.3 SymLexer

Nedan följer en beskrivning av hur SymLexer, vår lexer, fungerar. Lexern tar mot sym-koden den ska parsa som en sträng i sin konstruktor, denna sparas sedan undan och gås igenom steg för steg.

Lexern består av många delar, men de två viktigaste är:

1. En lång lista av regler.

Denna är en lista med reguljära uttryck parade ihop med ett Proc-objekt (ett slacks kodblock). Uttrycket bestämmer vad som matchas, och kodblocket bestämmer vad som ska hända om uttrycket matchar.

2. get_next_lexem() - funktionen.

Denna går igenom hela listan, letar efter något som matchar och om något matchar kör den kodblocket som är associerat med det uttrycket. I vissa fall är detta blocket odefinierat, för sådant vi vill ignorera. Exempelvis kommentarer ska inte lexas utan då fortsätter istället lexern med nästa rad utan att behöva involvera parsern.

Pseudokod:

Stega genom listan med regler, där det reguljära uttrycket är re och blocket block:

Om re.matchar(kod):

return block.call()

Kan vi gå igenom hela listan utan att matcha något är det ett syntaxfel och ett felmeddelande skrivs ut.

Bild 3.3.1 visar ett exempel hur lexern utför sin matchning. I detta fallet är det en ”ModifierOperator” den ska hitta. En modifier operator är exempelvis += operatorn. D.v.s. den ska modifiera vänsterledet med värdet från högerledet.

Listan med regler är som sagt uppbyggd i par, dels ett reguljärt uttryck och dels ett kodblock.

Den riktiga listan innehåller massor av regler, av pedagogiska skäl drar vi här dem till två regler, varav den andra är den som matchar.

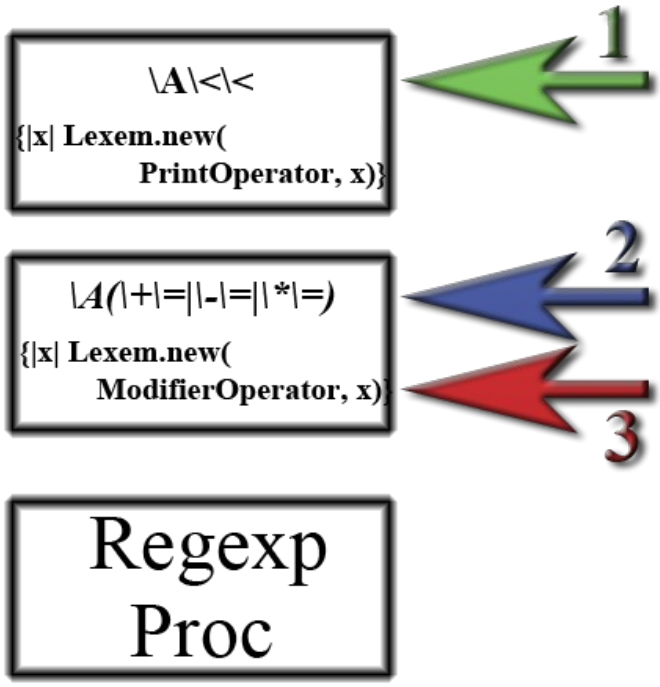

Bild 3.3.1 *Kodmatchning i SymLexer*

Stegen som visas i bilden är:

1. Matchar detta regexp? Nej det gör det inte, då går vi vidare till nästa kodblock.

2. Matchar detta regexp? Ja, gå till steg 3.

3. Kör kodblocket som tillhör detta regexp och returnera dess värde till parsern.

3.4 Scope och lagring

Vi hade inga större problem med att implementera en grundläggande form av scope. Sättet vi lagrar allt på är enkelt men funktionellt. Vi använder en scope-klass som vi skapar flertalet objekt av. Det skapas till exempel ett nytt scope när man går in i en funktion, och den tas bort igen när man går ur funktionen.

Alla dessa objekt ligger i en array i parsern (@scopes i koden). Det ska även poängteras att allt lagras tillsammans, i själva scope-objektet lagras i sin tur allt i en hashmap som namn=>objektreferens par.
Varje scope-objekt har 3 funktioner:

- Hämta namnet om jag vet objektreferensen
- Hämta objektreferensen från namnet (vanligast)
- Spara en ny objektreferens med ett visst namn

Vill man slå upp något, en variabel, funktion eller annat, söker man helt enkelt igenom denna array av scope-objekt och frågar varje objekt. Dessa objekt ligger naturligt så att den sista i arrayen lades på sist, därmed är det den ”mest lokala”. Eftersom vi söker igenom arrayen bakifrån och fram får vi därmed alltid en lokal variabel före en global.

Detta kan även skapa problem ibland. Exempelvis följande kod:

```
x = 1
$/fun1(Int x)
     fun2()
\$
$/fun2()
     << x
\$
fun1(2)
```

Detta kommer skriva ut 2, detta eftersom varje enskilt scope läggs i samma ordning som funktioner anropas. Detta är dock något av ett extremfall och bör inte ställa till några bekymmer så länge man är medveten om det.

3.5 Funktioner

Funktioner var egentligen den största och svåraste individuella biten att implementera. Dess implementation har gått igenom många iterationer innan vi tillslut beslutade oss för dess nuvarande utseende. Varje funktion sparar följande saker:

- Namn – Namn på funktionen, som sträng
- Returtyp – Självförklarande
- Kod – Sym-kod, som sträng. Mer om hur koden används nedan
- Argument – Som hashmap, nycklar=namn, värden=datatyp

OBS: Ifall returtypen eller argumentens returtyp är satt till Void tvingas ingen datatyp.

Ett separat funktionsobjekt skapas för varje enskild funktion i Symkoden. Denna sparas i ett scope-objekt (se 3.4.1) under funktionens namn parat ihop med en referens till objektet beskrivet ovan.

När man väl hittat objektet för den funktion som ska anropas tas följande steg:

1. Skapa ett nytt scope-objekt
2. Läs igenom argumenten till funktionen, matcha varje argument till ett värde i anropet och lägg in dem i det scope-objekt som skapades i steg 1 och lägg till det sist i scope-arrayen
3. Anropa parse() i parsern med koden i funktionsobjektet som argument. Detta gör att parsern skapar en ny instans av SymLexer, som får som uppgift att tolka den nyfunna Symkoden. Men spara kvar den gamla instansen, denna ska återställas senare

   Parsern gör samma sak som för all kod, d.v.s. evaluerar statements tills ett end of file-lexem tas mot (se avsnitt 3.2 för detaljer)
4. Parsern returnerar från parse() funktionen och vi finner oss åter igen på det ställe i koden vi befann oss. Den nya instansen av SymLexer tas automatiskt bort och byts tillbaka mot den gamla i slutet av parse()-funktionen.
5. Ta bort scope-objektet som lades till i steg 2
6. Parsern fortsätter automatiskt där den befann sig.

Steg 3 befinner sig alltså i parse()-funktionen, det vill säga samma funktion som startade exekveringen från första början. SymParsern är alltså rekursiv på detta sätt eftersom den anropar sig själv igen när den behöver evaluera en specifik mängd kod.

3.6 Klasser, objekt och metoder

När ovanstående två delar var klara (scope och funktioner) var det ingen större utmaning att implementera klasser, objekt och dess metoder. Detta eftersom en klass egentligen bara är en mall med ett något mer specifikt scope. Vi hade dessutom gjort det ganska lätt för oss eftersom vi inte hade stöd för publika variabler, och inte stöd för privata funktioner i grammatiken. Detta var ett medvetet val vi gjort redan från början eftersom det blir avsevärt mycket enklare att implementera. Vi hade redan ett tillräckligt tidskrävande projekt som det var.

Den slutgiltiga implementationen av dessa blev ganska enkel, avsevärt mycket enklare än vad vi kunde tro var möjligt. Koden har dock gått igenom många faser. De första försöken till implementation fungerade oftast i normalfall, men hade avsevärda problem med buggar och fick därmed skrivas om ett flertal gånger.

Klasser är implementerade som en mall varav man skapar objekt. Varje klass (obs, inte objekt) innehåller följande komponenter:

- Namn – Klassens namn, som sträng
- Variabler – Lista med variabler, behövs för att kunna skapa instans
- Metoder – Lista med funktionsobjekt, se 3.4.2

Observera att det alltså inte finns ett specifikt objekt för metoder, utan de återanvänder funktionsobjekten.

Objekt (d.v.s. instanser av klasser) lagras internt med följande information:

- Namn – Objektets namn, som sträng
- Variabler – Lista med variabler. OBS dessa modifieras inte, utan identifierar endast vilka som finns, detta behövs internt för vissa operationer

- Metoder – Lista med metoder.
- Objektscope – Ett scope-objekt, detta är objektets scope, det är här dess faktiska variabler och metoder lagras.

Vill man anropa en metod tas följande steg:

1. Referensen till objektet letas upp.
2. Objektets scope läggs på i parserns scope-lista (som referens, inte kopia).
3. Parsern utför ett funktionsanrop på metodens namn. Koden för funktionsanrop återanvänds alltså för att göra anrop till metoder.
4. Stegen beskrivna i 3.4.2 utförs.
5. Objektets scope tas bort från parserns scope lista.

3.7 Särskilda variabler, konstanter och metoder

Särskilda variabler, konstanter och metoder var inte något som var planerat från början, men vi kände att vi gärna ville implementera vissa ”extra” funktioner. Därav kom dessa till, här följer en (kort) lista på saker som särskiljs från mängden:

- NULL – Konstant, definierad som nil i implementationen.
- SYM_VERSION – Konstant, flyttal, versionsnummer för den version av Sym man använder, satt till 1.0 just nu, men kommer ökas ifall vi uppdaterar koden.
- sym_max_recursion – Variabel, heltal, tillåter användaren att bestämma maximala rekursionsdjupet för sin kod. Sätter man den till -1 blir rekursionsdjupet oändligt. Satt till 100 som standard.
- sym_current_recursion – Variabel, heltal, nuvarande rekursionsdjupet (främst till för debug-utskrifter).
- self – Variabel, referens till det objekt man är i just nu. Endast tillgänglig under exekvering av en metod.
- init() - Metod, konstruktor för objektet, denna måste vara definierad.
- __print__() – Metod, anropas ifall man försöker skriva ut ett objekt.

Nedan följer ett exempel på hur några av dem kan användas:

```
sym_max_recursion = 10
@/Person
     $/init()
          << self
     \$
     $/__print__()
          << sym_current_recursion
          << self
     \$
\@
p = Person.init()
```

3.8 Nästlad kod i loopar

Läsa in och hantera nästlad kod för repetitions- och villkorssatser löstes med att läsa in nästlad kod korrekt från en if-sats eller en loop.

Varje gång som parsern stöter på början av en sådan konstruktion går den in i dess motsvarande funktion i parsern – t.ex. om den hittar en början på en for-loop går den in i iterative_for_statement.

Vi kommer här använda for-loopen som exempel för att demonstrera hur problemet med att läsa in nästlad kod korrekt löstes.

Låt oss säga att vi har den här koden:

```
../ 1 10 i
     ../ 1 5 j
          << i
          << j
     \..
     << 1
     << 2
\..
```

Den nästlade koden innanför den yttersta for-loopen är en inre for-loop. Problemet som uppstår är hur parsern ska veta var ett kodblock slutar. Om den läser in all kod till nästa förekomst av \.. (end-for) kommer den läsa in:

```
../ 1 5 j
     << i
     << j
\..
```

Men kommer helt ignorera << 1 och << 2. Om vi däremot skulle läsa in all kod skulle den läsa in hela programmet, dvs. även kod som inte är en del av detta for-block.

För att komma runt detta problem behöver vi ta tillvara ett koncept om nivåer, dvs. att varje gång parsern stöter på ../ (början på en for-loop) ökar den nivån med ett. Därefter minskar den nivån varje gång den stöter på \.. (slutet på en for-loop). Därmed kommer nivån att förr eller senare komma tillbaka till noll, förutsatt att antalet början på for-loopar matchas av antalet slut på for-loopar. I andra fall skrivs ett felmeddelande ut.

Parsern behöver alltså rekonstruera koden i ett block och sedan skicka det till funktionen parse så att det kan parsas korrekt. Den rekonstruerar koden korrekt genom att anropa get_code_body() som är en del av lexern. Till denna metod anges som argument ett reguljärt uttryck med teckensekvenser som skall avbryta kodupphämtningen. get_code_body() returnerar därefter den resulterande koden, villkorsvärdet och nästa förekommande teckensekvens (next_part) i koden.

När parsern går in i iterative_for_statement ökar den for-nivån med ett. Därefter går den in i en for-loop och stannar kvar där tills for-nivån är tillbaka till noll. Den börjar med att hämta ../ 1 5 j. Då den hittat en nästlad for-loop så ökar den for-nivån med ett. Villkoret sparas som 1 5 j. Nästa del är ../ (begin-for).

I nästa skede av inläsningen av kod läser den in << i << j som kod och returnerar \.. (end-for) som nästa del. Eftersom det är en end-for minskar den nivån med ett. For-nivån är nu ett. Eftersom kvarvarande kod är \.. (end-for) kommer for-nivån minskas ytterligare med ett, och for-nivån är tillbaka till noll. Därmed avslutas while-loopen och den rekonstruerade koden parsas.

3.9 Nästlad kod i funktioner och metoder

Nästling av kod i funktioner och metoder löstes på ett lite annorlunda sätt. Detta främst eftersom projektet hade två utvecklare, och därmed att funktionerna utvecklades separat.

Nedan följer pseudukod för matchning av vad som ska ingå i en funktion. Observera att detta endast ger en pedagogisk överblick och är inte menat som ett komplett program. Den exakta koden ligger för den intresserade i metoden get_function_body() i filen lexer.rb.

```
Regexp = .*\$
startsymboler = 0

Upprepa:
     matchdata = Kör regexp på koden och spara det som matchar
     antal = Räkna antalet startsymboler ($/) i matchdata

     Om antal == startsymboler:
          Bryt ur loopen
     Annars:
          startsymboler = antal
          Generera ett nytt reguljärt uttryck som matchar det
          förväntade nästlingsdjupet
```

Huvuddelen som får detta att fungera är alltså att vi dynamiskt genererar ett reguljärt uttryck basera på det som matchas. Exempelvis i koden

```
$/fun1()
     $/fun2()
     \$
\$
```

Kommer först matcha den markerade texten:

```
$/fun1()
     $/fun2()
     \$
\$
```

Varvid ett nytt reguljärt uttryck kommer genererat, baserat på den startsymbol som hittades mitt i matchningen. Det nya uttrycket som genereras kommer nu alltså matcha hela fun1. Inga nya startsymboler hittades, och därmed kommer loopen brytas ur och koden returneras.

4 Utvärdering

Utvärderingen är baserad på de erfarenheter och lärdomar vi har dragit under projektets gång. Här försöker vi granska vad vi är nöjda med och vad som kunde gjorts bättre, samt vad vi tyckte om projektet och kursen som helhet.

4.1 Diskussion

Som en del av examinationen ombads vi ange cirka 20 synpunkter kring språk och programkonstruktioner, dessa hittas dels under rubriken ”diskussion” i Jespers reflektion, och dels under Nils observationer men vissa synpunkter finns även under övriga rubriker. Vi ombads också hålla en dagbok under projektets gång, denna är bifogad som bilaga och synpunkterna är till stor del baserade på de inlägg som finns under den. Men även baserade på dagboken som hölls under den tidigare kursen, TDP007, där vi studerade Ruby som programspråk och fick en introduktion till hur ett programspråk är uppbyggt.

4.2 Jesper, personlig reflektion

4.2.1 Inledning

Den huvudsakliga uppgiften med denna kurs har varit att skapa ett tekniskt språk, t.ex. ett programmeringsspråk. Genomgående har vi haft ganska stora friheter vad det gäller både språket och den slutgiltiga implementationen. Endast några enstaka begränsningar har gjorts på projektet, det måste vara implementerat i Ruby och vissa mindre detaljer.
Kursen har delats upp i två faser, först en planeringsfas där vi planlägger vilken typ av språk vi vill konstruera och diktar ihop grammatik samt annan dokumentation. Den andra fasen består av att göra en implementation av språket, d.v.s ett program som klarar av att köra ett stycke kod skrivit efter den grammatik som skrevs i planeringsfasen.

I slutskedet av kursen kommer vi även att lämna över implementationen och dokumentationen till en annan grupp på IP-programmet, de kommer då granska vårt projekt samt ge kritik vad vi kan förbättra. Detta ser jag som en särskilt viktig del då det ger utrymme att lära sig mycket som man säkerligen har nytta av framöver.

4.2.2 Projektet

Under planeringsfasen bestämde vi oss tidigt för en ganska hög ambitionsnivå. Språket vi valde att implementera är ett traditionellt imperativt och objektorienterat språk, fast med en tämligen minimal syntax, ett designval som vi anser ger språket både hög läslighet och hög skrivbarhet när man vant sig vid våra konstruktioner.

Ett viktigt beslut, som vi tog redan innan vi börjat implementationen, är att göra hela implementationen utan att använda några verktyg. Detta lyfter svårighetsgraden, och därmed ambitionsnivån, avsevärt. Dock anser vi att det är värt det eftersom vi då troligen kommer få ut avsevärt mycket mer av projektet. Använder man ett verktyg har man egentligen inte lärt sig så mycket om programspråk, utan snarare bara lärt sig använda ett färdigt verktyg (vilket i och för sig inte behöver vara dåligt). Orsaken till att man studerar är trots allt för att lära sig sådant man troligen inte hade kunnat lära sig på egen hand.

Vi har även vissa designval som är ganska ovanliga, exempelvis använder vi dynamisk typkontroll för vanliga variabler och liknande, men tillåter (inte tvingar) argumenten och returvärdet till en funktion att tvingas vara en viss typ. Detta har vi främst för att få en kombination av smidighet och pålitlighet, vi anser att det är upp till användaren att välja huruvida argumenten (och returvärde) ska tvingas vara en viss datatyp eller inte. För kritiska operationer vill man troligen ha den pålitlighet som statisk typning innebär, men för vissa delar vill man ha smidigheten av dynamisk typning. Detta visade sig emellertid vara ganska tidskrävande, mer än förväntat. Nu i efterhand känns många detaljer uppenbara som skapade avsevärda problem tidigare. Ändå är det inte ett val jag ångrar, jag anser att jag fått ut mycket kunskap av projektet, mycket av denna kommer jag ha stor användning för i framtiden.

Vår implementation är en interpretator som läser in hela filer i taget, skriven i programspråket Ruby. Ett språk som var ganska enkelt att lära sig, särskilt då jag har stor erfarenhet av Python, ett språk som har stora likheter med Ruby. Hade vi haft större friheter vad det gäller detta val hade jag dock valt ett kompilerat språk, som C++. Främst för prestandans skull, men också för att man har mer kontroll över detaljer, till exempel är det svårt att hålla koll på vad som är referenser och vad som är kopior i Ruby.

4.2.3 Grupparbete

För att projektet skulle bli rimligt utförs arbetet i grupper om två, jag har samarbetat med Nils Karlsson, en annan student på IP-programmet. Genomgående har grupparbetet fungerat bra.

Ansatsen av projektet handlade till stor del om att få upp en god struktur. Eftersom denna del handlade mer om teori snarare än att skriva mycket kod gjorde vi detta tillsammans. Vi träffades i de datorsalar som tillhör IP1 och implementerade strukturen till den nivå då vi kände att den var tillräckligt generell för att senare kunna utökas med samtlig funktionalitet.
Därefter skedde stor del av arbetet hemma, vi använde oss till stor del av SVN via min privata server för revisionshantering, vilket gjorde att vi kunde arbeta parallellt med separata delar av programmet i varsin branch som sedan sammanfogades. Därmed lyckades vi dela upp arbetet bra och arbetsbördan blev så rättvist fördelad som är möjligt.

Vi höll ständigt kontakten via email, chat(MSN) och träffades på universitetet vid behov. Till följd av det lyckades vi hålla varandra uppdaterade med vad vi implementerade. Efter varje uppdatering läste vi även igenom varandras kod, det är lätt att bli hemmablind i sin egen kod. På det sättet höll vi oss också uppdaterade med hur alla delar av programmet fungerade.

4.2.4 Diskussion

Denna del innehåller främst diskussion kring konstruktioner och uppbyggnad av programspråk. I första hand baserad på den språkdagbok som hölls under detta projekt samt motsvarande dagbok från TDP007-kursen.
Jag personligen har bakgrund i flertalet språk, både från tidigare kurser på LiU och gymnasiet, men jag har även studerat flera språk på fritiden. Därmed anser jag mig ha en ganska god bas att stå på, och har provat på flertalet språk. Jag har åtminstone grundläggande kunskaper i: C/C++, Assembler, Visual Basic, PHP, HTML/XHTML, Lisp, Python, Ruby och Perl. Jag har även provat på vissa mer exotiska språk som exempelvis Prolog, men har ingen direkt insikt i hur de fungerar.
Normalt brukar man också vara starkt påverkad av det första språket man lärde sig. För mig var det Visual Basic, ett språk som jag inte alls använder längre.

De flesta saker som ofta diskuteras inom programspråk verkar nästan alltid komma ner till en avvägning mellan hur kraftfullt språket blir för en van programmerare mot hur lätt språket är att lära sig. Min personliga åsikt är att kraftfulla språk generellt är bättre, åtminstone för icke-triviala tillämpningar. Ett språk som exempelvis Visual Basic är grymt bra för att lära sig hur program fungerar samt för mindre, triviala program. Men ifall man överväger att göra ett större program i VB bör man starkt överväga andra alternativ.
Dock är detta som sagt fortfarande en övervägning som bör göras. Det finns även vissa språk som är så komplicerade, och därmed svårlästa, att de blir svåra att använda för riktigt stora program. Ett bra exempel på detta är Lisp. Mycket trevligt språk när man förstått hur det fungerar, men om jag gör ett större program skulle jag helt klart föredra ett traditionellt, objektorienterat, språk som C++.

En annan indelning man skulle kunna göra är hur lätt ett språk är att skriva i, jämfört med hur lätt det är att läsa. Jag anser dock att dessa två helt klart går att kombinera, exempelvis Python anser jag vara både väldigt läsbart och lätt att skriva i. Lätt att skriva därför att språket är dynamiskt typat, har ett avsevärt standardbibliotek samt lätt att läsa eftersom man är tvungen att indentera korrekt och därmed att koden flödar ganska naturligt.
Om man behöver göra ett val, exempelvis vid konstruktion av ett eget språk, skulle jag säga att läsbarhet går före skrivbarhet. Att kunna göra uppdateringar på koden ett halvår efter man skrivit den är viktigare än att spara in lite tid när man håller på att skriva. åtminstone för icke-triviala program.

En konstruktion jag har vetat om länge, men som jag inte använt i praktiken förrän i början av TDP007 kursen, och använt i större skala vid detta projekt är reguljära uttryck. Kortfattat är de en beskrivning av ett mönster, som man sedan kan göra vad man vill med. Vi använder det i vår implementation för att matcha konstruktioner och känna igen varje del i grammatiken. Det tog avsevärda ansträngningar att lära sig hur dessa används, i början hade jag stora svårigheter att matcha mer än triviala saker. Men efter mycket experimenterande kläckte jag idén och tycker idag att dem är mycket smidiga. Dessa är något jag helt klart kommer använda ofta i framtiden. Dock är de främst smidiga för den som skriver dem. Alla icke-triviala uttryck blir väldigt svårlästa för någon som läser de i efterhand.
Exempelvis ett uttryck från vår implementation:

```
\A\w+\(( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)?(,( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)*( )*\)
```

Knappast lättläst.
Vilket leder till nästa del, kommentarer. Kommentarer är ovärderliga vad det gäller att förtydliga programkod. Ovanstående uttryck hade nästan varit oläsligt, om man inte lägger avsevärd tid på att tyda uttrycket, om det inte vore för kommentarer. I vår kod har vi försökt kommentera alla svårlästa bitar, exempelvis hör kommentaren att det är ett funktionsanrop som matchas till uttryck ovan.

Såhär i efterhand hade jag gärna implementerat projektet i ett annat språk än Ruby. Ruby är ganska smidigt, i och med dess dynamiska typning. Men jag hade sparat in avsevärd tid genom att använda ett statiskt typat språk, exempelvis C++. Även om dynamisk typning är smidigt förlorar man mycket av den kontroll som statisk typning innebär. Det är svårt att hålla koll på vad som blir referenser och vad som blir kopior. Man skulle emellertid kunna hävda att detta att detta är okunskap om hur Ruby fungerar, vilket delvis är sant. Ändå anser jag att man i ett dynamiskt typat språk inte har samma koll på detaljer som man skulle ha i ett statiskt språk.

I många språk, bland annat Ruby och vårt eget språk, finns det vissa specialfunktioner. I Ruby finns exempelvis många metoder man kan definiera i sina klasser som implementerar vissa specialfall, exempelvis method_missing som anropas om en viss metod inte kunde hittas i ett objekt. Ett

exempel i vårt språk en två speciella, globala, variabler som heter sym_max_recursion och sym_current_recursion. Som namnen antyder tillåter den första att man kan sätta det maximalt tillåtna rekursionsdjupet, den andra vilket rekursionsdjup man ligger på just nu. Ett annat exempel som finns i många språk, bland annat vårt, är ”self”, en referens som pekar på det objekt koden körs i för närvarande.
Personligen är jag något kluven ifall dessa konstruktioner är bra eller inte. Det är helt klart snyggare med ett väldigt ”generellt” språk som endast har ett par grundläggande regler som sedan allting följer, utan specialfall. Många gånger kan detta dock visa sig vara något av en utopi om man inte är beredd att offra många andra saker som gör språket svårare att använda. Dessa konstruktioner för helt klart till en viss dynamik till språket. Överlag skulle jag säga att det är bra, så länge det inte går till överdrift.

En annan sak man bör betrakta när man skapar ett språk är hur förlåtande språket ska vara. Ska man till exempel tillåta en ”else” gren före en ”else if”? I vårt språk är det tillåtet, och villkoren evalueras fortfarande korrekt. Att ha ett väldigt hårt och oförlåtande språk kan ge all kod ett något mer enhetligt utseende, vilket ökar läsbarheten, men det gör det svårare att skriva kod i det för nybörjare. Min åsikt är dock att språket bör i denna bemärkelse inte tvinga vissa saker allt för hårt. Vill man ha snygg och läsbar kod bör man få det genom uppsatta standarder för hur koden ska se ut, hur koden ska kommenteras och liknande, inte vara en begränsning i verktyget.

Finns det då ett bästa språk, för alla? Nej, åtminstone inte än på länge. En bra programmerare bör välja ett programmeringsspråk efter den uppgift han ska utföra, inte hålla sig fast vid ett specifikt bara för att det är det man lärde sig först.
Det finns många avväganden man bör beakta, är det ett stort eller litet projekt? Krävs plattformsoberoende? Finns det några prestandakrav? Måste bakåtkompabilitet hållas med ett tidigare projekt?
Först när man svarat på bl.a. detta, är man beredd att välja ett språk för just den tillämpning man tänkt sig.

4.2.5 Slutsats

Jag anser att projektet har varit mycket lärorikt. Att konstruera ett språk ger helt klart bättre förståelse för hur ett program byggs upp. Därmed anser jag att det är något som varje programmerare bör erfara, särskilt om man inte använder några verktyg.
Att skriva grammatiken kändes särskilt nyttigt då detta var något helt nytt för mig. I början kändes det emellertid ganska besvärligt. Framförallt hade jag problem med notationen samt att få överblick hur samtliga saker hängde ihop. Efter att ha studerat grammatik för språk man redan kan(Python) blev det dock betydligt lättare. När man väl börjat inse hur allt hänger ihop var det inga större problem.

Ett fåtal delar av vår ursprungliga språkspecifikation visade sig vara ganska svårimplementerade, exempelvis att instansvariabler skulle använda en statisk datatyp. Detta visade sig vara svårt framförallt eftersom vi kombinerade det ihop med dynamisk typning för vanliga variabler. Hade vi gått igenom med den idén hade vi behövt lagra alla klassvariabler separat. Dessutom har vi ändå stöd för att kontrollera in-argumenten till metoder, vilket i de allra flesta fall borde räcka som typkoll. Grammatiken är bifogad som bilaga, samtliga ändringar är gjorda i den.

Hade jag gjort om projektet från början idag hade jag troligen inte ändrat något. Projektet har varit mycket lärorikt, även om man har fått skriva om vissa kodstycken flera gånger allt eftersom man hittar nya problem. Även om det är irriterande att göra om saker är det lärorikt, förhoppningsvis gör

man inte samma fel i framtiden.
Jag antar att syftet med kursen var att ge en bättre förståelse för programspråk. I så fall anser jag helt klart att detta mål är uppnått. Resultatet av projektet, en fungerande implementation av ett eget programspråk, är också något jag är nöjd med och något man kan känna sig stolt över. Även om det säkerligen finns kvar buggar och förbättringar som skulle kunna göras.

4.3 Nils, Personlig reflektion

Redan under ett tidigt skede i projektets gång bestämde vi oss för att implementera en egen parser och lexer, dvs. att inte använda de redan färdiga konstruktionerna som fanns tillgängliga i rdparse och utgå från dem. Vi visste att det skulle bli betydligt mer utmanande och tidskrävande, men även mer givande i form av de kunskaper vi förvärvar. Utifrån detta har vi lärt oss en hel del om hur språk är konstruerade och hur designval kan påverka ett språk i dess helhet. Här tänkte jag berätta lite om hur jag upplevde arbetsprocessen från dess att vi skrev en grammatik av språket till att vi implementerade den grammatiken.

Större delen av den första perioden planerade vi språket – vilket inkluderar dess grundläggande grammatik och hur de olika konstruktionerna kan användas.

Vi började skriva grammatiken för språket med tanken om att syntaxen helst ska begränsas till specialsymboler som finns tillgängliga på tangentbordet. Detta gäller särskilt för konstruktioner som funktioner, klasser, loopar och if-satser, som i många språk utgörs av ord. Därmed gav vi språket till sist namnet "Sym" – en förkortning av *Symbol*.

Att komma igång var nästan inga problem. I och med att vi skulle implementera en egen parser och lexer visste vi inte riktigt var vi skulle börja. Men när vi väl hade satt oss in i hur en s.k. "*recursive descent*" parser är uppbyggd (varje grundläggande regel i grammatiken motsvaras av en funktion i parsern) kunde vi nästan genast sätta igång att implementera språket.

En stor del av tiden i början av projektet ägnades till att få parsern och lexern att samarbeta så att vi kunde koncentrera oss på problemdomänen, dvs. varje funktion i parsern. Det första var att få den lexikaliska analysen att fungera korrekt. Detta innebar att få lexern att dela upp en sekvens av tecken till s.k. "tokens" korrekt.

Denna separering av en sekvens av tecken till tokens gjordes som förväntat med s.k. reguljära uttryck, som bäst kan beskrivas som ett kraftfullt sätt att matcha en sträng av text mot ett mönster. Även om jag har läst lite om det tidigare har jag inte använt det i samma skala som i det här projektet. Ett av de viktigaste momenten med att konstruera lexern var att bemästra reguljära uttryck, vilket jag också känner att jag har gjort. Det är en teknik som jag säkerligen kommer ha användning av i framtiden, i alla fall som ett alternativ till andra liknande tekniker, så som DOM och SAX.

När vi väl hade lexern färdig var nästa steg att implementera parsern, som förstås anropar lexern och utifrån de genererade enheterna ("tokens") har en sidoeffekt, t.ex. att text skrivs ut eller att en variabel tilldelas ett värde. En parser tolkar m a o enheterna och utifrån denna tolkning har sideffekter.

Det mest utmanande momentet i denna del av projektet var att sluta tänka linjärt och istället börja tänka rekursivt, vilket inte alltid är en intuitiv process. Detta i och med att funktionen parse (som parsar ett givet kodblock) anropar sig själv med nästlad kod.

Språket implementerades i Ruby. Detta hade vissa för- och nackdelar. Den största fördelen är att Ruby har många konstruktioner och är ett väldigt koncist och flexibelt språk. Nackdelen med Ruby är att det inte är särskilt lämpat för att implementera något så prestandakrävande som ett eget språk. Om jag fick möjligheten att skapa språket på nytt skulle jag välja ett statiskt typat språk med stöd för objektorientering, såsom Java eller C++.

Ett av de huvudsakliga syftena med denna kurs var att lära sig hur ett språk är uppbyggt och vilka olika val man har att göra med avseende på sådant som berör språkets syntax och liknande. Ett exempel som kan nämnas är räckvidd, dvs. var en variabel, funktion, klass eller någon annan konstruktion som kan kommas åt med ett namn är tillgänglig. T.ex. hade vi att en variabel som deklareras som en del av en for-loop endast är tillgänglig innanför for-loopen, att den har en räckvidd som omfattar for-blocket, men inte mera.

Det gick bra att jobba i projekt. Det blev mycket individuellt arbete, men med kontakt via e-mail, msn och andra kommunikationskanaler. De första dagarna när vi ville komma igång med arbetet jobbade vi gemensamt på universitetet, för att sedan dela upp arbetet för att på så sätt hinna implementera så mycket som möjligt. Under projektets gång träffades vi även på universitetet för att diskutera vissa aspekter av språket och för att kombinera det vi hade gjort.

Sammanfattningsvis gick jag in med lite kunskap om hur vi skulle implementera språket till att större delar av det vi har gjort numera känns väldigt uppenbart. Den främsta tröskeln var att till att börja med att få parsern att fungera med lexern. Därefter stötte jag personligen på några utmanande aspekter av implementeringen. Jag behövde utveckla ett koncept av nivåer för loopar och if-satser för att på så sätt konstruera koden korrekt i ett givet kodblock. Detta kan utan tvekan sägas vara intellektuellt stimulerande, då det kräver mer problemlösning och kreativitet kring hur man kan använda de olika konstruktionerna i ett språk på bästa möjliga sätt.

Till slut några ord om projektet jämfört med tidigare projekt. Detta projekt var mycket mer utmanande. Att jag är van vid den traditionella Java- och C++-syntaxen gjorde det inte lättare att komma igång. Att lära sig det extremt flexibla språket Ruby var en tröskel att komma över. Det känns väldigt givande att ha implementerat ett eget språk. Det är definitivt något jag kommer fortsätta jobba på efter kursen är avslutad.

4.4 Nils observerar 10 saker utifrån utvecklingsblogg

Utvecklingsbloggen används som underlag för de observationer vi har kunnat ta del av under projektets gång. Även om jag har observerat en hel del så tar jag här upp de viktigaste observationerna, som för mig hade mer genomslagskraft.

1. **Variablers räckvidd i loopar**

I de flesta språk finns loopar där man definierar en variabel som en del av loopen, t.ex. for (int i = 0; i < 10; i++), där variabeln i fungerar som en stegningsvariabel. Ett designval var alltså hur lång räckvidd denna variabel har och ifall den är *muterbar*, dvs. att man kan hoppa till slutet av loopen genom att sätta i till 10 i själva for-blocket, så här:

```
for (int i = 0; i < 10; i++) {
     i = 0;
}
```

I Java sätter den i till 0 vilket gör att den hamnar i en oändlig loop. Variabeln i har endast räckvidd i dess relaterade for-block. I Sym valde vi att göra variabeln i omuterbar, dvs. att det inte går att ändra dess värde. Den har – precis som i Java – en räckvidd endast inom dess associerade for-block. Anledningen till att variabeln i är omuterbar är att det inte finns någon anledning att ändra dess värde. Det ger upphov till dålig, svårläst och ostrukturerad kod. Det går att ändra på dess värde innanför loopen, men den sätter alltid dess värde till nästa stegningsvärde vid nästa iteration:

```
../ 1 3 i
     << i
     i = 3     # Tillåtet
     << i
\..
```

Detta ger utskriften: 1, 3, 2, 3, 3, 3

2. **Sanningsvärden – vad är Sant och Falskt?**

En annan sak som skiljer sig åt i olika språk är vad för värden som anses vara Sanna och vilka som anses vara Falska. Är en tom sträng Sann eller Falsk? En sträng med text då? En sträng som innehåller text av ett tomt värde då, t.ex. "null" eller "void"? Ja, det finns inte något objektivt måttstock (som i matematiken) för att komma fram till detta. Vi har implementerat detta med att sätta 0 till ett falskt värde medan resterande värden anses vara sanna, även negativa tal.

3. **Att vara eller inte vara tillåtet. Vad accepterar programspråket som indata?**

Att ett språk är flexibelt har både för- och nackdelar. Fördelen är att man lätt kan få saker gjort. Nackdelen är att man för lätt kan få saker gjort, vilket kan resultera i konstiga buggar. I Ruby är det t.ex. tillåtet att skriva:

```
"1".upto("10") {|x| puts x}
```

Detta skriver ut: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Sedan har vi t.ex.

```
"5".upto("10") {|x| puts x}
```

Vilket inte skriver ut något, men returnerar "5"(!) Buggen kunde uppstå då jag glömde konvertera från sträng till heltal. Detta är ett exempel på för mycket flexibilitet och ett behov av typsäkerhet.

4. **Strikt ordning eller... är det okej att vi ställer oss lite hur som helst?**

Jag känner inte till något språk där ordningen i en if-sats kan ändras, med avseende på else if och else. Först kommer If, därefter Else Ifs, till sist Else. Denna strikta ordning har vi dock inte med i Sym, utan istället kan man byta plats på Else och Else Ifs, så här:

```
?/ 1 == 2
       << 1
\*             # Else
       << 2
\?/ 2 == 2     # Else If
       << 3
\?/ 3 == 3     # Else If
       << 4
\?
```

Dock går det här att köra korrekt. Det är en lustig detalj som får Sym att skilja sig åt från andra, även om det är möjligt att det kan anses vara förvirrande.

5. **Konsekventa programkonstruktioner**

Då Sym är baserat på specialsymboler som är tillgängliga på tangentbordet uppstod frågan om

operatoröverlagring – har vi tillräckligt med symboler för att använda i språket? Måste dessa överlagras, dvs. att en symbol kan användas på olika sätt, beroende på i vilket sammanhang (eng. *Context*) som den befinner sig i? Det uppstod inte direkt några problem med detta, annat än att vi i vissa fall inte haft någon symbol som reflekterar dess funktionalitet. Exemplet kan tas med "return" som vi använder för att returnera ett värde från en funktion. En konsekvent lösning vore förstås att låta samtliga nyckelord vara symboler, men det finns ingen bra symbol för "return". Möjligtvis -> som dock kan blandas ihop med C++, där den är en så kallad "dereference"-operator. Ett annat förslag på att lösa detta var att helt enkelt returnera det sista värdet i funktionen, ungefär som i Ruby. Detta är dock en "osynlig" konstruktion – nackdelen är att man inte ser direkt vad som returneras. Än så länge använder vi oss av return för att returnera värden.

6. **Listor och hashes**

Frågan om listor och hashes uppstod – nämligen vad som får vara tillåtet i en lista respektive en hash. Är det möjligt att spara uttryck i en lista som därefter evalueras när de hämtas? I hashes, vad för element är giltiga som nycklar respektive värden? Går det att spara en given lista som nyckel? Många frågor uppstod och det blev oerhört komplext. Hur dessa implementeras korrekt syntaxsmässigt bidrog också till komplexiteten. Än så länge har vi inte implementerat dessa konstruktioner. Annars anser vi att listor och hashes skall vara så flexibla som möjligt, dvs. att de accepterar vilken indata som helst egentligen, som kan evalueras till ett värde. Särskilt då vi inte har några restriktioner på typer. I Java som är typrestriktivt måste man t.ex. specificera vad för typ av element som får finnas i en array, även om det är klassen Object, som är superklass till alla subklasser.

7. **Är det tillåtet att skriva ut en uträkning?**

I Ruby är det tillåtet att skriva:

```
x = 2
puts x *= 2     # => 4
puts x          # => 4
```

Detta kan liknas vid Java där det är tillåtet att skriva:

```
System.out.println("x: " + ++x)
```

På grund av vår parsers natur valde vi inte att implementera uträkningar av uttryck vid utskrift. Exempelvis går inte detta:

```
x = 0
<< x *= 2
```

Detta ger upphov till en SYM-ERROR av slaget "*Invalid statement*".

8. **Läsbarhet, Skrivbarhet och pålitlighet**

Man måste göra vissa avväganden mellan läsbarhet och skrivbarhet i språk. Java är t.ex. väldigt verbost då det krävs en hel del för att bara skriva ut "Hello World". Om detta kan göras på 10 rader i Java kan det göras på 1 rad i de flesta dynamiska språk, såsom Ruby och Python. Detsamma gäller för Sym. Desto mer komplex programkoden blir desto mer svårläst blir den i Ruby och Python – medan i Java är det väldigt enkelt att läsa sig till vad den betyder, då det är verbost och förlitar sig mer på engelska än specialsymboler. Sym är inte skrivet för att vara läsbart, utan för att vara skrivbart. Detta då det förlitar sig till stor del på specialsymboler. Detta gäller även för

konstruktioner som i andra språk består av engelska ord, som exempelvis if, class, function, etc. Dessa definieras i Sym med specialsymbolerna ?/, @/, $/. Sym lutar sig alltså mer mot skrivbarhet än läsbarhet. Att lära sig skriva dessa symboler korrekt i början kan vara svårt, men efter ett tag vänjer man sig vid denna syntax. Pålitligheten hänger starkt samman med vissa faktorer, bland annat läsbarheten, men även i vad språket i dess helhet accepterar som indata.

9. **Är konstanter konstanta eller konstanta så länge du vill att de ska vara det?**

En överraskande detalj i Ruby är att det går att ändra värden på en konstant efter att den har antagit ett värde. Så, varför kallas det för en konstant då? Jo, för att en varning visas när man ändrar på dess värde. I de flesta andra språk är konstanter konstanta. Ruby är skrivet med tanken att det ska vara flexibelt. Frågan är om programmet skall avslutas eller visa en varning och fortsätta. En vanlig programmeringsprincip är att avsluta så fort som möjligt, ifall ett fel uppstår. Detta borde utan tvekan även vara sant för konstanter. Därför har vi implementerat det så att programmet avbryts:

```
Constant = 123      # OK
<< Constant         # OK
Constant = 324      # INTE OK!
```

Den sista raden ger upphov till en SYM-ERROR med meddelandet *"Constant CONSTANT already in use in this scope*".

10. **Operatorprioritet**

Till sist tänkte jag prata lite om operatorprioritet. De aritmetiska operatorerna har en egentligen i förväg bestämd prioritet, då de finns tillgängliga i matematiken. Uttrycket 1 + 2 * 3 evalueras t.ex. till 7, enligt tanken om att operatorn * har högre prioritet än 1. Det är egentligen inget konstigt med detta, med tanke på att multiplikation egentligen är flera additionsoperationer. M a o betyder det - om man bryter ned det till dess beståndsdelar – 1 + 2 + 2 + 2 = 7. Men frågan är hur man ska bestämma prioritet med avseende på operatorer som inte är aritmetiska, utan som är en del av ett programspråk? Det är inte lika givet hur vissa operatorer skall prioriteras framför andra. Prioriteten i sig kan ställa till det då den kan göra koden svårläst och betyda olika saker beroende på hur de är prioriterade. Ett sätt att komma runt detta är m h a parenteser. Detta används ofta för logiska operatorer (|| och && i Sym eller or och and i Ruby) för att förtydliga prioriteringen, även om parenteserna i sig inte har någon effekt på hur de operatorerna prioriteras. Trots detta är operatorprioritet en bra sak. Särskilt gäller detta för aritmetiska operationer – vilket är de vanligaste operationerna i ett språk.

5 Bilagor

Del 5 innehåller flertalet bilagor till dokumentationen. Bland annat finns här teknisk grammatik för språket, flera kodexempel samt programkoden för implementationen.

5.1 Grammatik

Kort förklaring av den notation som används i grammatiken:

En eller fler	+
Noll eller fler	*
Noll eller en	?
Ett tecken	.
Eller	|
Literal (del av syntax i SYM, inte grammatiken)	Inom ” ”, alternativt prefix \
Regel	STORA BOKSTÄVER: Definition
Kommentar	Påbörjas med //

```
PROGRAM             : COMPOUNDSTATEMENT

COMPOUNDSTATEMENT   : {STATEMENT}+

STATEMENT           : BUILTINFUNCTION EXPRESSION
                    | PRIMARY
                    | EXPRESSION

PRIMARY             : "(" COMPOUNDSTATEMENT ")"
                    | CONDITION
                    | CLASS_DEF
                    | FUN_DEF
                    | WHILE_LOOP
                    | FOR_LOOP
                    | ASSIGNMENT
                    | RETURN_STATEMENT
                    | BREAK_STATEMENT

EXPRESSION          : SINGLEOPERATOR EXPRESSION
                    | EXPRESSION OPERATOR EXPRESSION
                    | FUN_CALL
                    | METHOD_CALL
                    | IDENTIFIER
                    | DIGIT
                    | STRING
                    | DICT
                    | LIST

ASSIGNMENT          : IDENTIFIER "=" EXPRESSION

DATATYPE            : Int                       // Obs stor bokstav, sen små
                    | Long                      // För inbyggda datatyper
                    | String
                    | Float
                    | Bool
                    | Void

CONSTANT            : A-Z, {A-Z}+               // Automatiskt konstant vid stora bokstäver

IDENTIFIER          : a-z, {A-Z | a-z | DIGIT}+

NONZERODIGIT        : 1-9, {0-9}
```

```
DIGIT                   : 0-9, {0-9}

NONNEGATIVE_INTEGER     : NONZERODIGIT | 0

COMMENT                 : "#".*?TERMINATOR

MULTILINECOMMENT        : "#/".*"\#"

STRING                  : \".* \"

RELATION                : EXPRESSION":"EXPRESSION

DICT                    : "{"(RELATION(","RELATION)*)?"}"                       // {}, {x:x}, {x:x, y:y}

LIST                    : "["(EXPRESSION(","EXPRESSION)*)?"]"                   //  [], [x], [x,y]

OPERATOR                : +
                        | -
                        | *
                        | /
                        | ==
                        | !=
                        | %
                        | ** | ^
                        | >=
                        | <=
                        | >
                        | <
                        | "||"
                        | "&&"

SINGLEOPERATOR          : !                                                     // Endast en operand

BUILTINFUNCTION         : <<                                                    // print med newline
                        | <<<                                                   // print utan newline
                        | >>                                                    // indata


CONDITION               : "?/" EXPRESSION TERMINATOR                            // Villkor
                            COMPOUNDSTATEMENT
                            (
                            { "\?/" EXPRESSION TERMINATOR                       // Else if
                                    COMPOUNDSTATEMENT
                            }+
                            |
                            [ "\*/" TERMINATOR                                  // Else
                                    COMPOUNDSTATEMENT
                            ]?
                            )
                        "\?"

CLASS_DEF               : "@/" IDENTIFIER TERMINATOR                            // Klass
                            (IDENTIFIER TERMINATOR)                             // Instansvariabler
                            (FUN_DEF)*                                          // Metoder
                         "\@"

FUN_DEF                 : "$/" DATATYPE? IDENTIFIER "(" {DATATYPE? IDENTIFIER ,?} ")" TERMINATOR
                            COMPOUNDSTATEMENT
                         "\$"


WHILE_LOOP              : "|/" EXPRESSION TERMINATOR
                            COMPOUNDSTATEMENT
                         "\|"


FOR_LOOP                : "../" (NONNEGATIVE_INTEGER | IDENTIFIER) (NONNEGATIVE_INTEGER | IDENTIFIER)
                            [NONNEGATIVE_INTEGER | IDENTIFIER]
                            COMPOUNDSTATEMENT
                        "\.. "

RETURN_STATEMENT        : return EXPRESSION

BREAK_STATEMENT         : <--
```

TERMINATOR : "\n"

CALL_ARGS : (EXPRESSION(","EXPRESSION)*)?

FUN_CALL : IDENTIFIER"("CALL_ARGS")"

METHOD_CALL : IDENTIFIER.IDENTIFIER"("CALL_ARGS")"

5.2 Kodexempel

Denna del innehåller flera kodexempel som illustrerar hur de olika konstruktionerna i språket kan användas.

5.2.1 Klasser 1

```
@/Person
    name
    age
    $/init(String arg_name, Int arg_age)
            name = arg_name
            age = arg_age
            << name
            << age
    \$

    $/__print__()
        <<< "Namn: "
        << name
        <<< "Ålder:"
        << age
    \$
\@

obj = Person.init("Jesper", 21)
<< obj

# Skriv ut 10 ggr
../1 10 i
    << obj
    << "------"
\..
```

5.2.2 Klasser 2

```
@/Person
    name
    msg
    $/init(arg_name, String arg_msg)
        name = arg_name
        msg = arg_msg
    \$

    $/__print__()
        <<< name
        <<< " says: "
        << msg
    \$

    $/String get_msg()
        return msg
    \$
\@

x = Person.init("Zombiekid", "I like turtles")
<< x
x_msg = x.get_msg()
<< x_msg
```

5.2.3 Dictionaries, relationslistor

```
x = {"darkside":"cookies", {"test":{1.0 : 2}}:2,
"goodside":"cake"}

<<< "Darkside has "
<< x["darkside"]
```

```
cakeowner = "goodside"
<<< "Goodside has "
<< x[cakeowner]

<<< "What side do you join[goodside/darkside] ? "
>> choice
<<< "Here, have some "
<< x[choice]
```

5.2.4 Funktioner och rekursionsdjup

```
sym_max_recursion = 50
$/funk()
    << sym_current_recursion
    funk()
\$

funk()
```

5.2.5 Iteration

```
list = [1, 2, 1+2, 2*2, 5, "6", 14/2, (2**2)*2]

../0 7 i
    << list[i]
\..
```

5.3 Språkdagbok

I denna del har vi kopierat in den dagbok vi skrev efter de flesta arbetstillfällen. Här har vi skrivit lite kort om de problem som har uppstått, hur vi tänkte och löste dom.

```
Onsdag 24 Feb:
* Påbörjad implementation av lex-analys
Inte generell ännu, men en bra början.



Måndag 22 Mars:
* Planering



Tisdag 23 Mars:
* Vi skrotade den tidigare koden vi skrivit för att göra en mer generell kod.
* Vi har pÅbörjat en ny implementation som istället använder reguljära uttryck
m.m för att analysera koden.
Detta gör implementationen mycket enklare att utöka med ny funktionalitet. Vi
har även separerat pÅ lexern och
parsern istället för att ha dom ihopbyggda som i den tidigare implementationen.



Onsdag 24 Mars:
* Strukturerat om stora delar av koden med hjälp av boken samt trial and error.
* Till imorgon läser vi igenom vÅr kod nogrannt och besÅ vi kan fortsätta vÅr
implementation
* Just nu kan vi parsa expression och returnera deras värden, samma som igÅr
fast nu pÅ ett betydligt snyggare sätt.



Torsdag 25 Mars:
* Lade till stöd för paranteser
* Utskriftsoperatorn
* Stöd för flera rader
* Fixade bugg med -/+
* Fixade bugg med return_previous
* PÅbörjat utskrift utav textliteraler, ÅterstÅr dock endel arbete (buggar)
* Lade in en möjligt debug_mode flagga
* Flytta datatyps-deklarationerna till en separat fil
* Rensade upp strukturen



Fredag 26 Mars:
* Lade till stöd för en del operatorer
* Inmatning
* Tilldela och slå upp variabler
* Skrev en todo-lista
```

* Gjorde en större testfil
* Konstanter
* Negativa heltal
* Grand previous lexem
* Diverse buggfixar

Nils, Lördag 27 Mars (arbete hemma)
* Division by zero. Visar ruby fel. Förmodligen har vi väl raise här för att skriva ut DivisionByZero och på vilken rad.
  Ruby skriver ut DivisionByZero men om vi ska skriva ut på vilken rad felet sker på så vore det bra att ha en sådan raise.
* Hittade en bugg i parser.rb i metoden assignment_operator(). Se raden:
      raise "Error: Invalid lexem for assignment #{@varname.type}"
  Detta bör vara:
      raise "Error: Invalid lexem for assignment #{varname.type}"
  Alltså, utan @.
* Lade till <, >, >= och <=.
* Vid jämförelse körs: factor1 = factor1.to_i > factor2.to_i
  Detta då när factor1 och/eller factor2 är variabler sparas de som strängar. Man kanske kan spara de som lexems på något sätt...
  Men hur det skulle gå till är jag inte hundra på. Ska tänka lite på det.
* Tänkte strax börja implementera lite enklare sym-syntax i jEdit. Får se hur det går.

Nils, Söndag 28 Mars (arbete hemma)
* Är i princip färdig med syntax highlighting och indentering för sym-filer i jEdit. Skapade en katalogstruktur modes/jedit där
  sym.xml finns. När jEdit är installerat finns en mapp modes där alla xml-filer ligger. Sen är det bara att gå in på
  Utilities > Troubleshooting > Reload Edit Modes.
  Det jag mest funderar på är var whitespace får och får inte förekomma. T ex
      @/Person
  Highlightas även om det är ett whitespace efter Person, men endast ett - två och flera dödar highlightingen.
  Inbyggda datatyper har grön färg.

Nils, Onsdag 31 Mars (arbete hemma)
* Är det tillåtet att skriva:
      << x *= 2
I ruby är det tillåtet att skriva:
      puts x *= 2
Som det ser ut just nu fungerar det ej.
* Skapade två metoder get_variable() och get_constant().
* Lade in operatorerna *=, /=, -=, +=, etc. etc.

Jesper, vecka 13(arbete hemma):
Har varit hemma hos föräldrarna under större delen av veckan och har därmed inte arbetat så mycket. Följande har dock hänt:

* Slå upp vars/consts med funktion och säkerhetskoll, annars riskerar man nilvärden och knas
* Skriver ut lite mellanrum i början om debugmode är på, så man ser enklare i en termial var utdatan börjar
* Småkommentarer lite här och var
* Hashbang i början av sym.rb, Nu körs programmet i en terminal genom "./sym.rb testfil.sym"
* Flyttade L/R-Paranthesis sist i regexpen, behlövddes för att få funkdef att fungera
* Grundläggande stöd för funtioner
* \n-regexpet matchar nu en eller flera nyrader, så flera blankrader kan termineras i en omgång av parsning. Bättre prestanda och enklare utdata vid debug_mode
* Fixade till många debug-utskrifter samt raise-fel (ändrade också endel puts "error..." till raise istället så programmet dör)

Funderingar:
* Kunna spara egna datatyper(klasser) i funktionernas parameterlista
* Skriva testfall, pipe:a till fil som man sedan kollar att varje rad stämmer med utdata?
* Radnummer i felmeddelanden

* Massa regexp måste vara i viss ordning i listan, annars kan dom krocka med varandra, t.ex skapar "Hej=5" en konstant med namn H och en variabel med namn ej

* Funktioner returnerar hur?
* Operatorer/funktioner för att omvandla mellan datatyper? T.ex läsa in ett heltal:
```
>> x
x = x->Int
# x är nu en int
```

Läser in string, som omvandlar till int?

Nils, Söndag 4 April (arbete hemma)
* Lade till ; som Terminator.
* Det enda alfabetiska nyckelordet vi har är "return" vilket är inkonsekvent. För datatyper har vi att de är alfabetiska
  och för att denotera funktioner, klasser etc. har vi specialtecken. Det kan vara bra att ha ett specialtecken även för
  return. Frågan är då vad? Det enda jag kan tänka mig är ->
* Byta namn på metoden get_numbers(). Till vad?
* Exakt hur ska vår implementation av Listor och Hashes se ut?
* Smått påbörjat implementera listor, men har problem med rekursitivitet. T.ex. [[[]]], ergo listor i listor. Den tillhandahållna
  lösningen finns dock som list_container_handler()

Jesper, Tisdag 6 April(arbete hemma):
* Gjorde en ändring i print_operator-funktionen. Om debug mode är påslaget highlightar den utdatan som sym-programmet skriver ut för att den inte ska begravas i debug-infon.
* Började kolla på hur scope ska implementeras. Skapade en ny klass och fil för det. Tanken är att parsern håller en array med flera Scope-objekt, och den som ligger sist i arrayen är "mer lokal". Dvs vid ett funktionsanrop lägger vi på ett nytt scope-objekt i slutet när funktionen anropas, i den lägger vi även in

argumenten som den ska anropas med, koden anropas och sedan tas det scope:et bort ur arrayen och koden fortsätter som vanligt.
* Hittade bugg med multiline-comments om en funktionsdef ligger i den blir det fel. Får kolla upp senare, troligen något problem med regexp:en som krockar.
* Skapade mappstruktur i min branch, så testfilerna ligger i en egen mapp och backend-källkoden i en egen mapp. Enklare för en "användare" om det bara finns en huvudfil.
* Lade till en funktion scope_trace() i parsern, den skriver ut _ALLA_ scopes i ordningen "mest lokal" och uppåt, mest till för debug-syften för att kunna se vad som har sparats och inte.
* Scope fungerar som det bör nu

Funderingar:
* Hur ska vi (syntax/grammatik-mässigt) definiera vilket scope man anser? I en funktion kanske man vill ändra på det globala scopet, eller i scopet för objektet om man är i en metod osv...?
* Ska man kunna definiera en ny konstant i ett mer lokalt scope? Just nu kan man det. Ex om det finns en konstant PI i globalt scope, ska man då få skapa en lokal PI? Osäker hur det är i andra språk...
* I huvud(main)-filen borde vi kolla om det finns en fil att ta mot i argumentlistan, skickar man inget argument ny så bara sitter den och väntar.
* Vi bör spara radnummer, för felmeddelanden. Kan göras i parsern ganska enkelt, interne variabel som räknas upp när man stöter på en terminator (OBS, Terminator godkänns på flera ställen, även i funktions-def)

Komma ihåg att testa:
* Nästlade funktioner
* Funktionsanropinn från en funktion
* Scopes

Jesper, Onsdag 7 April(arrofbete hemma):
Har lagt mesta tiden på att skriva om sättet funktioner definieras. Mycket snyggare och generellt bättre nu, förra sättet hade även endel begränsningar som riskerade att de gammla regexpen krockade samt andra fel.
Nu behövs inte heller fun_parts-fulvariabeln och function-def klassen sköter inte massa arbete med ful-split. Dessa var dessutom hårdkodade till bara ett argument.

Har också lagt till try/catch (eller som dom heter i ruby: begin/rescue) i mainfilen, den kommer fråga om man vill läsa debug ifall man inte gav flaggan via kommandoraden.

Mindre viktiga ändringar:
* Lexern har fått en ny funktion code_left() som hämtar all kod kvar UTAN massa whitespace som skräpar ner debug koden
Ex skrivs nu:
```
Lexer: Getting a lexem code left='$/ Int plopp(Int a) << a \$ PI = 3 PI = 3.14'
```

Och innan:
```
Lexer: Getting a lexem code left='$/ Int plopp(Int a)
  << a
\$
PI = 3
PI = 3.14'
```

Blir lättare att läsa debug utan massa överflödig kod :)

* Tog bort lexem.rb filen, den är bara några rader så den får lätt plats i

datatypes.rb istället.
* Lade till en "DataType"-typ, för att identifiera när man skriver Int, Float osv..
* Lade till en funktion som genererar regexp'n istället för i konstruktorn, detta för att vi ska kunna lägga till tillåtna datatyper sedan. (Annars hårdkodas de till Int, Float osv...)
* Mellanslag/Tab-regexp matchar nu flera i taget.
* Skriver nu ut var spårutskrifterna kommer ifrån (lexer/parser) i debug mode
* Lade en till endel alias för vissa saker som inte finns i ruby. Ex skapade Int som alias för Fixnum etc...
* Ändrade så terminator-lexemens värde är "NewLine" istället för "\n", förstörde debug utdatan lite
* Lade till globala konstanter NULL och NIL

Ändrade två saker i huvudfilen(sym.rb):
* Tog bort require lexer. Räcker med parsern (den inkluderar i sin tur lexern...)
* Ändrade om hur argument till sym fungerar. Nu tar den även -debug flaggan för att enkelt byta till debug mode i hela projektet
Lätt att lägga till både om man kör vi kommandorad och enkelt att sätta 2 olika configs i en IDE(netbeans/eclipse, en med och en utan debug-flaggan)
Nu är det bara köra "./sym.rb Test/test_source.sym -debug"

Funderingar:
* Tillåta anders förslag? Specifiera en funktion i funk-defs argumentlista som validerar om argumentet är giltligt? Borde inte vara allt för svårt... (Tillåta funktion på något sätt i regexpet /syntax, sen evaluera den funktionen med argumentet och om returdatan == false raise:ar vi något)
* Ska vi ha några try/catch statements?
* Namnet evaluate_statement är lite missvisande i vissa situationer då det används för att räkna ut expressions
* Skapa ett faktiskt fel som heter SYM-ERROR och raise:a det istället?
* Funktionsanrop med 2 argument fungerar fortfarande inte. Den går definiera men inte anropa. Det blir problem med regexp:et då den tror plopp(5,3) är variabeln plopp ELLER funktionsdef beroende på hur man lägger regexpen. Går säkert lösa på något smart sätt..

Viktiga:
* Lade även märke till att exempel 2 i får språkspec är lite konstig, konstruktorn sätter annat än de variabler vi skickar med. Personnummer istället för namn...
* Är ruby hashmaps i samma ordning som man lade in saker? I python är dom inte det. Är dom inte det här kan det skapa obekväma buggar då många saker lagras så. T.ex funktionsargumenten kan komma i fel ordning i så fall.

Nils, Torsdag 8 April (arbete hemma)
* Börjat implementera if-satser. Funktionen conditional_statement sköter detta. I funktionen skiljer den mellan if, else if
  och else, istället för i lexern (som bara tar ut ett helt if-block). Nästlade if-strukturer har inte blivit testade ännu.
  Villkor i if-satserna har inte blivit implementerade ännu. Detta behöver fortfarande göras.

Nils, Söndag 11 April (arbete hemma)

* Den första idén om IF-satser visade sig vara alltför svår att genomföra. Istället kom jag på ett elegantare sätt att lösa det:
  istället för att läsa in ett helt if-block så läser lexern in enskilda if-delar, t.ex. IfPart, ElseIfPart, ElsePart, etc. etc.
  Utifrån dessa finns en räknare (@if_level) som räknar vilken nivå if-satsen befinner sig på. Varje gång programmet kommer in
  i en ny if-sats räknar den upp. När en if-sats avslutas räknas den ner med ett. conditional_statement() fortsätter köra så
  länge @if_level inte är nere på noll, dvs. så länge det finns flera if-delar att hämta innanför ett givet if-block.

  Om ett villkor evalueras till sant (e.g. 1 == 1) hämtas all kod till nästa if-del. Sedan körs denna bit kod genom att anropa
  metoden parse och sätta fun_call till true för att indikera att vi endast vill köra detta kodblock.

  Programmet måste också läsa tokens från if-delar där villkoret har evaluerats till falskt. Dessa tokens kastas sedan iväg och
  skickas alltså inte till metoden parse. Istället sätter den en variabel ignore_next till true för att indikera att metoden parse
  inte ska parsa detta ogiltiga kodblock.

  Om ett villkor evalueras till falskt (e.g. 1 == 0) hämtas nästa lexem. Detta lexem bör vara en if-del (felkontroll finns inte
  här för tillfället). I alla fall anropas conditional_statement() förutom vid EndIf, där räknaren går i minus med ett.

Nils, Tisdag 13 April (arbete hemma)
* Har strukturerat om i conditional_statement() så att det blir mindre utskrifter (inte mer än nödvändigt) och skapat en funktion
  is_conditional() för att kolla om en lexem är en If-part.
* Har börjat implementera for-loopar. Metoden för att hantera dessa är iterative_for_statement(). Jag passade också på att skapa en
  metod som tar in kod tills ett villkor evalueras till falskt. Metoden heter get_code() och tar som argument en funktion. Denna
  funktion anropas för att evaluera villkoret. Koden + lexemet som avbröt upptagningen av koden returneras.

14 April:
Handledning med Anders och pratade lite efter om vad vi gjort, vad som är kvar, buggar m.m.

Jesper, 14-15 April (arbete hemma):
Började implementera klasser
Skrev om slutet på class_def(), nu returnerar den det man behöver för att skapa funktionen, istället för att skapa den direkt. På detta sätt hoppas jag att man kan återanvända den i klasser.

Skrev om debug-meddelandet för när ett regexp matchar, nu skriver den ut vad den matchade på (t.ex ClassDef, Variable etc). Lite mer lättläst än att få regexp:et den matchade på
Lade till så debug i lexern inte skriver ut om det är en terminator, tar upp onödigt med uttrymme. Samt att den inte skriver ut när den matchar newlines/kommentarer längre,

Nils, Torsdag 15 April (arbete hemma)

* När vi läser in reguljära uttryck i lexern bör vi ha en metod add_regexp(regexp, datatype) för att göra detta. Vi har rätt mycket duplicerad kod där. Vissa block har sidoeffekter på den inläste datan, t.ex. att den tar bort citationstecken eller liknande. Kan vi göra detta någon annanstans? Detta är refaktorisering. Vi har egentligen viktigare grejer att prioritera, men om vi har tid över kan vi strukturera om lite.

Jesper, 18 April (arbete hemma):
Har inte blivit så mycket arbete under helgen, har mest studerat matte.

Kom eventuellt på ett sätt hur man kan få - operatorn att fungera utan mellanslag efter, och ändå klara negativa tal:
Man har bara en gemensam för dom, som returnerar värdet efter sig fast invers, alltid. Och ha typ en "default operator", dvs plus, alltid räknas på det tidigare om inget angivet. Ex:
5 -2 skulle skapa ett heltal -2, och eftersom ingen operator är angiven antas plus, så den räknar 5 + -2 snarare än vad man tror.
Har inte börjat implementera ändringen, ska kolla vidare på klasser som jag har endel bekymmer med.

Jesper, 19+20 Apil (arbete hemma):
Dessa kommentarer skrevs under längre arbetspass med endel avbrott, vissa av noteringenarna kan därför vara motsägelsefulla då dom inte är skrivna i ordning =).

Har tänkt om hur klasser ska implementeras och fick skriva om en del, men tror det blir bättre på detta sättet i längden.
Om man inte anger datatyper (ex init i klasser) så defaultas det till void.
Fick skriva om assignment operator något

Vi behöver kunna returneras saker ur funktioner/metoder och använda det på ställen som expressions, ska kolla på det imorgon kanske.

Ändrade felmeddelandet i lexern ifall ingeting matchade, innan sa den hur många tecken från slutet som va kvar, säger inte så mycket så nu skriver den ut koden den kommit till istället.
Tog bort funcall argumentet till parser konstruktorn
Vi borde kanske lagra alla variabler som int/float internt? Verkar som vi bara använder string nu.
Satte så alla klassvariabler initeras till 0
Skippar datatyper på klassvariabler, så länge iaf, blir betydligt jobbigare att kolla det varje gång

Lade till endel varningar, ger inte fel men skriver ut en varning. Ex om man omdefinierar en klass i samma scope
Finns fortfarande bugg med datatyperna i funktioner/klasser, den tror Int argument ska vara string.
Skrev ihop Variable och Constant i evaluate statement, de ska innehålla samma kod, det är upp till scope/storage att se till att constant inte skrivs över.
Borde kanske skriva omdel saker i scope klassen så dom blir lite enhetliga senare om det finns tid över.

Ska vi tillåta defaultargument i funktioner / metoder?
Ska objekt kunna referera sig själv? En inbyggd konstant "self" eller liknande som går att nå inne i metoders scope. Borde gå att lägga till ganska enkelt, när man gör funktionsanropet skapar vi ändå ett scope för argument, där kan man

lägga in en konstant eller variabel med referens till sig själv, borde gå utan problem.
Gjorde endel småändringar här och var, orkar inte skriva alla =)

Skrev om seek_storage lite, nu kan man skicka med en tredje flagga som gör att den inte raisar fel ifall den inte hittar det man letar efter, då returnerar den false istället. Detta för att man ska kunna kolla om en variabel finns, detta behövs i assignmentoperator för att få scope att fungera ordentligt. Innan bara lade vi till variabeln i "mest lokala" scopet, men om det redan finns en variabel/whatever definierad i ett högre scope ska vi istälelt ändra på den istället för att skapa en ny lokal variabel. Detta skapade problem för mig med klassvariabler, därför ändringen skedde.
Lade till en global variabel @return_level i parser, denna håller i vilket scope något hittades. Kom inte på något bättre sätt att veta var en viss sak finns. Problemet ligger i att man måste kunna slå upp klassvariabler i högre scope, och i så fall ändra denna istället, man måste då veta i vilket scope det ska skrivas över.

Kom på en sak med hur scope är definierat. Man borde nu (har inte orkat prova) kunna ha en klass som heter x, en variabel som heter x, en konstant som heter x OCH en funktion som heter x i samma scope. Borde kanske inte vara så, scope borde kanske internt lagra alla ihop.

Annan sak: Har inte provat ännu (går inte än), men potentiell bugg finns med scope i objekten, det sparade scopet uppdateras bara lokalt under metodens körning och sedan görs en kopia på den (görs inte ännu, men sen). Gör man då ett metodanrop på samma objekt kommer den inte få alla lokala ändringar som gjorts hittils. Exempel:

```
@/Person
      personNR
      $/init(Int p_nr)
                personNR = p_nr
                print()
      \$

      $/Int print()
                << personNR
      \$
\@

x = Person.init(5)
```

Kommer nu skriva ut 0, efterson det är det som klassvariablerna initieras till. Den ändring man gjorde i init kommer inte kopieras in i objektets kopia av scopet förrens dess funktion har tagit slut.

En till notering är att vid funktionsanrop m.m så skickar vi alltid argumenten som variabler, oavsett om de uppfyller de syntaxmässiga kraven för att egentligen vara konstanter, kan skapa problem då den kommer försöka slå upp på const och det kommer inte fungera. Ex om argumentet är PI och man sedan kör << PI kommer det inte gå, för den sparar argument som variabel, men koden slår sedan upp efter en konstant.

Jesper, 22 April (arbete hemma):
* Började skriva på automatiserade testfall som provar alla konstruktioner i koden och som man kan köra och se så man inte bryter något vid ändringar
Skapade nya mappar för autotestsen med massa filer i, en fil för varje "gren" i testen (funktioner, uttryck etc..)
Lägger till en flagga i parsern som hindrar utskrifter (silent mode).

Testfallen beter sig mycket underligt och jag har haft endel problem med dom. Exakt samma testfil som körs fungerar om man bara kör den rakt upp och ner, men kör man den genom ett unittest fungerar vår lexer inte (fastän den får in själva texten, det har jag kollat).
Ett annat problem va att den inte lyckades importera våra filer om man gjorde det globalt, man va tvungen att göra det i en funktion, mycket underligt.

* Började skriva på endel testfall som INTE ska gå igenom, tänkte att vi sammlar dom i en fil så vi kan köra alla sedan.
* Ändrade på felmeddelandet i lexern ifall den inte har något regexp som matchar. Nu skriver en ut koden därifrån den är tills nästa nyrad, tycker detta blev ganska snyggt då det blir skriver ut var felet är:SYM-ERROR:
Invalid lexem at .change(-42)
Fastän att det finns mer kod under, tycker vi bör göra liknande på alla felmeddelanden.

* Lade till så man får ha stora bokstäver som i hela klassnamnet
* Fixade till debugutskrift när klasser skapas. Nu skriver den ut vad klassen den skapar har för namn, metoder, deras returtyp och eventuella argument till metoderna.

* Skrev om regexp för metod/funktions-anrop, nu matchar den allt som bör, utan att matcha annat.
Det nuvarande (och något svårlästa) regexpet för metodanrop är nu:
```
\..+\(( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)?(,( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)*( )*\)
```

Blev lite pillande innan jag fick till det, regepet är smidiga för att matcha, men är lite kluriga och icke-triaviala regexp blir hyffsat(läs mycket) svårlästa.
Som testfall till regexpet använde jag följande GILTLIGA anrop:
```
Person.init()
Person.init(5)
Person.init(5.0)
Person.init(5,3, 1.0,4)
Person.init(-5)
Person.init(-5,2, -2.0)
Person.init("hej")
Person.init(   "hej",  -2,        1)
Person.init("hej", -2, 1, "hej")
TH
```
Och ICKE giltliga som inte ska gå igenom:
```
Person.init(2,)
Person.init(-1,)
Person.init("hej",)
Person.init(2,,3)
Person.init(,)
Person.init(   "hej",  -2,
1)
```

Det fungerar bra och som förväntat. Liknande för vanliga funktionsanrop (utan punkt i början av regexp och samma i testfallen)
Jag använde ett inbyggt verktyg i netbeans (utvecklingsmiljön(IDE) jag använder) som liknar rubular.com något, dock hjälper den även till att strukturera upp regexpet i bitar på ett väldigt smidigt sätt samt markerar i realtid vad som matchar och inte.

Jesper, 24 April (arbete hemma):
Har börjat fundera på hur vi tvingar en viss datatyp på argumenten som skickas in, jag har stött på endel svårigheter med detta. Mer specifikt uppstår svårigheterna om man ska tvinga en egendefinierad datatyp, alltså en egen klass man har skapar. Hur kollar man att det man skickar med är ett objekt av den klassen när vår implementation av klasser egentligen inte är klasser utan bara lurar användaren till att tro det? (eget scope av variabler man inte kommer åt utifrån etc..)?
Jag föreslår därför att man kan tvinga inbyggda datatyper i argumentlistan genom att skriva Int, Flaot osv... Fast man kan också välja att inte specifiecera något, och då tvingas inget. Detta används då för anropa funktioner med "specialobjekt".
Alternativt så har man en funktion i sitt eget objekt som returerar vad den är för något, denna funktion kanske skapas automatiskt när man definierar klassen. Som helt enkelt returerar vad den är för något, t.ex skulle det kunna vara .class funktion på objektet som då returnerar strängen "person" eller liknande, därmed kan man kolla mot den. Detta kan dock bli något krångligare att implementera.

Har behövt skriva om en stor del av hur klasser definieras för att undvika fel i vissa extemfall, detta har tagit mesta av dagens tid. Problemet som uppstått är att man även måste kunna defnieria tomma funktioner och med det nya regexpet jag implementerade för funktionsanrop så matchas även funktionsdefinitioner som inte har argument, ex print(), och eftersom det inte finns något (enligt min uppfattning) enkelt sätt att gå runt detta måste istället koden för att defniera funktioner skrivas om så att den även accepterar detta lexem. Ifrån detta lexem:et kan man sedan i sin tur splitta ut (med hjälp av regexp) namnet eller ev argument. Jag planerar att då matcha alla funktionsdef även med denna och splitta ut det därifrån istället.
Hur jag ska få till syntaxcheck på detta är dock ett hittils ett mysterium. Man borde kunna splitta ut varje del i taget och kolla den efter ett särskilt regexp och se om det matchar.

Skrev om hur datatypes identifieras så att man ska kunna ha olika, t.ex en Int kan också vara en Fixnum eller en Integer.
Har haft endel problem med buggar. Det verkar som att argumentlistan till funktioner(och därmed också metoder eftersom den återanvänder funktionsdef-koden) alltid väljer den sista av dom möjliga uppräknade datatyperna.

Ska komma ihåg och prova om det går att anropa med objekt, det borde det göra, men inte säker.

Hittade även endel "buggar" med klasser eller rättare sagt scope. Är lite osäker på vad av detta som går göra i andra språk, men dom känns som dom borde finnas i endel av dom iaf:

```
@/Person
    p_nummer

    $/init(Int nr)
        p_nummer = nr
    \$

    $/Int print()
        << p_nummer
    \$

    $/Int call()
        jag.print()
    \$
\@
```

```
jag = Person.init(5)
jag.call()
```

Fungerar eftersom objektet jag är med i globalt scope

```
@/Person
    p_nummer

    $/init(Int nr)
        p_nummer = nr
    \$

    $/Int print()
        << p_nummer
        jag.print()
    \$

    $/Int call()
        jag.print()
    \$
\@

jag = Person.init(5)
jag.call()
```

Skapar en oändlig loop tills ruby säger ifrån att stack-minnet har tagit slut =)

Jesper, 26 April:
Viktig sak:
Regexpen/strukturen för metoder eller nått måste skrivas om, så som dom är nu krockar vissa saker med andra och jag får hela tiden flytta runt på regexpen för att välja vilken av dom som ska funka.
Lägger jag functioncall ovanför variables fungerar funktionsanrop OCH variabler, men inte metodanrop så som:
x = Person.init()
Måste fixas ASAP.

Tvärtom så fungerar inte funktionsanrop som statements, ex:
print()

En annan viktig sak:
Scope behövde arbetas om lite, nu kan man ha både objekt, variabler m.m med samma namn.
Ex:
x = Person.init()
<< x
Fungerar inte eftersom den förväntar sig att x är en variabel (och om stora bokstäver typ PI så förväntar sig den konstant)
Har nu skrivit om hela scope och refaktoriserat koden på alla ställen så den använder sig av de nya funktionerna. Nu lagras ALLT tillsammans i istället för en seperat som tidigare. Så slår man nu på x får man objektet som är lagrat under x, utan att specifikt veta om det är en variabel, ett objekt etc...
Fick därmed skriva om en hel del saker, bla SymObject klassen.

Fixade massa buggar med funktionsanrop, hoppas bara att jag inte skapade fler :D

Kan nu returnera värden ifrån funktioner, detta görs genom att sätta en instansvariabel i parsern när den hittar returnoperatorn

Lade även till en ny flagga i parsern, @break, som avslutar parsningen och återgår som ifall den hade slut på symkod. Detta används när ett funktionsanrop stöter på return.
Vi måste vara försiktiga med return, efter att scope har fixats ordentligt (se ovan) kanske man kan returnera saker i init-funktioner, detta kanske kan skapa problem.

Kom på hur man kanske kan kolla att ett objekt man får in är av en specifik datatyp:
När man letar upp argumenten i själva anropet så kollar man objektets "är av klasstyp" variabel mot vad funktionen förväntar sig, så lägger man ansvaret på funktionsanropet snarare än själva funktionen.

Jag provade köra följande:

```
@/Message
    msg
    $/init()
        msg = "I like turtles"
    \$
\@

obj = Message.init()
<< obj
```

Det skriver ut samma som ruby hade gjort, dvs <SymObject bla bla..> grejer.
Jag hade en idé att man kallar en funktion (säg __print__()) på objektet om man försöker skriva ut den

Fick bugg av:

```
@/Message
    msg
    $/init()
        msg = "I like turtles"
        << "Set msg to: " + msg
    \$
\@
obj = Message.init()
```

Står att den inte kan konkatenera fixnum och string, undersök vid senare tillfälle

Jesper, 29 April (arbete hemma):
Har skrivit om hur funktioner definieras så att de kan användas i varandra, t.ex:

```
$/print()
    << "gooooo print!"
    $/calc()
        << "Calculating"
    \$
    << "done calculating"
    calc()
\$

$/something()
    << "ERROR!!"
\$
```

```
print()
```

Funkade tidigare inte, men nu fungerar det.

Fixade också en allvarlig bugg som tidigare gjorde att antingen fungerade klasser eller funktionsanrop, detta va pga hur regexpen va placerade och man var tvungen att "välja" ett i taget. Detta fixade jag nu genom att skriva om FunctionCall-regexpet något så det förväntar sig ett antal word-characters vid ett anrop istället för .+

Hittade en bugg med inmatning, följande fungerar inte:

```
<< "x?"
>> x
```

Jesper, 1 Maj:
Inte arbetat så mycket(skriver tal + inte hemma ikväll), men observerade en mindre sak:
Har man långa kommentarer, t.ex 100 rader så tar det en bra stund innan parsern dör ut, den borde snarare matcha allt som en kommentar på en gång och bara dö ut.
I detta fall hade jag två flerrads-kommentarer, men det verkade som han körde en stund efter och gick igenom det iaf (minst 5 sekunder, men printade inget iaf)

Ett annat problem, detta fungerar i vanlig läge, men i debug mode får jag "undefined method `slice' for nil:NilClass". Kolla närmare på det sen:

```
$/poke()
    << "poked"
\$

poke()
<< x
```

Nils, Fredag 30 April (arbete hemma)
* Strukturerat om bland primaries. Istället för att loopar och if-satser läses in lexem för lexem hanteras det nu genom att man
  går in i själva koden och hämtar ut koden som text istället. På så sätt behöver vi inte återskapa lexemen när koden skall parsas
  (credits till Jesper som påpekade detta). While- och for-loopar fungerar i princip. Rekursivitet (loopar i loopar) fungerar också.
* Upptäckte ett fel i koden som måste fixas. Givet att vi har koden:

```
      ?/ 1 == 1
            << "\*"
      \?
```

  kommer den hantera * som ett lexem som skall tolkas som en ElsePart, fastän att det är en del av en sträng.

Nils, Tisdag 4 Maj (arbete hemma)
* If-satser är 99% färdigimplementerade. Det som fattas är felkontroll och genomförlig testning av if-satserna, för att se att de
  fungerar korrekt. Refaktorisering av koden kommer nog också behövas. Projektet börjar närma sig sitt slut.

Nils, Onsdag 5 Maj (arbete hemma)
* Lade till felkontroll till if-satser vid multipla else-satser i en "platt" if-struktur. Skriver även ut fel ifall en if-sats inte
  är avslutad korrekt.
* Fixade till sym.xml (sym edit mode till programmeringseditorn jEdit) så att den indenterar korrekt

Jesper, Lördag 8 Maj (arbete hemma)
* Har skrivit om hur objekt initieras och hur metoder anropas.
Detta pga att det var mycket repetativ kod i båda som redan fanns i function_call, det kändes mer rätt att återanvända koden som redan var skriven här.
Nu fungerar de båda så att de lägger in objektets scope, sedan gör ett funktionsanrop med namnet. Eftersom då objektets scope ligger sist kommer det alltid hittas först och det blir då det som anropas.

* Lite småfixar här och var, endel kommentarer m.m
* Arbetat med dokumentationen. Skrev början till systemdokumentationen samt ritade lite illustrativ grafik.

* Fixade allvarlig bugg med funktionsanrop som gjorde att argumenten inte tvingades korrekt. Tidigare lagrade vi inbyggda datatyper i en lista och itererade över dom. Nu skrivs regexp:en explicit vilket inte skapar buggen. Buggen uppkom troligen av att typen i vårt proc block sparades som en referens, vilket skrevs över i varje iteration, därmed skapade den ALLTID det som låg sist i listan.

* Skapade en funktion, get_line(), i lexern som hämtar ett "bra" kodstycke att använda som felmeddelande. Antingen allt som redan matchas på denna raden om det finns, annars allt som är kvar på denna raden.
* Gjort om stor del av funktionsdef koden, ingen snygg lösning, men buggar inte lika mycket nu.

* Skapade en global konstant SYM_V, satt till 1.0. Sym version :)
* Skapade en variabel "self" i objekt/klasser
* Har lagt till en global variabel, sym_max_recursion som bestämmer hur stort rekursionsdjupet max får vara
* Har även lagt in en sym_current_recursion
Ex:
```
sym_max_recursion = 5
@/Person
    msg
    $/init(String arg)
        msg = arg

        self.run()
    \$

    $/run()
        << msg
        << sym_current_recursion
        self.run()
    \$

    $/__print__()
```

```
        << msg
    \$
\@

zombiekid = Person.init("I like turtles")
```

* Gjorde multiline comment regexpet lazy, buggar annars
* Tog bort classname variabeln, den krockade med variable/constant ibland. Lade istället in motsvarande sak i classdef delen, som jag sedan matchar mot och slicar ut för att få klassnamnet.
* Lade till fler escapes till strängar space, \\, m.m
* Tog bort scope_trace funktionen, den var inte implementerad för senaste versionen ändå. Och nu är vi ändå snart klara så behöver inte debugfunktionen.
* Har löst problem med att metoder inte kunde ta mot variabler, objekt och liknande saker. Löste detta genom att först kolla om det finns definierat lagrat, isf använder man den, ANNARS evaluerar den literalen.

* Gjorde en <<< operator, den printar ut saker utan att infoga newline efter.

+ En hel hög grejer jag inte orkar skriva :P

Jesper, 9 Maj (arbete hemma):
* Arbetade massor med dokumentationen. Skrivit mesta av systemdokumentationen nu.
* Fixade en allvarlig och hyffsat exotisk bugg med returtyper. Pga alla de nästlade if-satserna i retur-funktionen var den svår att se, men efter lite provande lyckades jag återskapa den baserat på koden.
Buggen uppstod om man:
1. Tar in ett _INT_ argument
2. Försöker returnera denna varaibeln EFTER modifikation. Detta gör att den sparas internt som fixnum, inte Int. Dessa två är givetvis samma sak, men om man kör en == koll på dom misslyckas det.
Ex:

```
$/Int funk1(Int arg)
    return arg+1
\$
```

Detta löste jag genom att lägga till ett specialfall i både argument och retur som tillåter fixnum och Int att bytas om varandra och fortfarande fungera.

* Fixade en bugg som smygit sig in. Man kunde omdefiniera konstanter. Detta pga jag skrev om den koden men glömde skriva en todo för det tidigare, men löste det enkelt.
* Kommenterade lite av koden samt snyggade till.
* Fixade så man kan uttryckligen säga att något är Void, innan vad det bara implicit.

Nils, 12 Maj (arbete hemma):
* Noterar att @iterative_while_level, @if_level och @iterative_for_level inte längre behöver vara klassmedlemmar, utan kan vara lokala i deras respektive associerade metod.
* Hittat några buggar.
* Fixade ett fel med kod i conditional_statement(). Förut var det

```
      code_to_run = code_to_run.strip
```

  Numera är det:

```
      code_to_run = code_to_run.strip if code_to_run
```
Då code_to_run initieras med värdet nil så kan man alltså inte köra strip() på den. Då måste man lägga till ”if code_to_run” - vilket gör att den endast får ”strippas” när code_to_run _inte_ är nil.
* Lade till felutskrifter vid If, m.m. Förut skrev den ut ../
Numera skriver den ut ../ 1 10 i, dvs. ../ följt av villkoret.
* Lade till break-operatorn <--
Den kan endast användas innanför loopar. Försöker man ”breaka” utanför en loop får man en felutskrift. Problemet är dock att den inte hoppar ur loopar. Den går ut ur loopen. Med andra ord körs den resterande koden som kommer efter <--, vilket den inte borde göra. Fixa till detta.

Nils, 13 Maj (arbete hemma):
* Vid vissa felutskrifter (SYM-ERROR) ”strippar” den numera get_line() med strip() för att bli av med överflödig whitespace.
* Break operatorn <-- fungerar nu som den ska.
* Lade till stepping i for-loopar, vilket gör att man kan hoppa över ett fix antal tal i ett intervall.
* Ändrade i check_numeric_error(). Förut kollade den om klassen var Integer, Float, eller BigNumeric. Numera gör den en sådan jämförelse på en rad med Numeric === 1.
* Lade till en preliminär inkrementeringsoperator – fungerar endast med c++, och inte ++c. Kan endast användas i en statement. Endast variabler kan plussas på.

5.4 Programkod

Nedan följer samtlig kod skriven till implementationen. En översiktlig förklaring av koden kan hittas i systemdokumentationen.

5.4.1 lexer.rb

```
# -*- coding: utf-8 -*-
require 'Backend/datatypes'


class SymLexer
  attr_reader :sym_code

  def initialize(sym_code, debug_mode = false)
    @debug_mode = debug_mode
    @sym_code = sym_code
    @line_code = ""              # Stores the code from the current line

    # All custom datatypes (classes)
    @datatypes = []

    # If set to true, will return last lexem on the next get_nex_lexem()
call
    @return_previous = false
    @previous_lexem = nil        # 1 Step back
    @grand_previous_lexem = nil  # 2 Steps back
    generate_regexp()
  end

  # Gets the remaining code with excessive whitespace removed
  # This is used for debug output
  def code_left(code = @sym_code)
    re = /\s+/
    return code.gsub(re, " ").strip
  end

  # Returns an appropiate chunk of code to use in errors
  def get_line()
    return @line_code + /.*/.match(@sym_code)[0]
  end
```

```
  # Empties @line_code variable, (called when change line)
  def flush_line()
    @line_code = ""
  end

  def add_datatype(new_type)
    puts "Lexer: Adding new datatype #{new_type}" if @debug_mode
    @datatypes << new_type
    generate_regexp()
  end

  # Returns function body (also used for methods as they
  #                          are only functions in a confined scope)
  def get_function_body()
    total_re = //m
    any_re = '.*?'
    end_re = '\\\\' + '\$'
    begins = 0
    endloop = true

    result = @sym_code
    begin
      endloop = true
      # (any + (end+any)*begins) + end
      total_re = Regexp.new(any_re + (end_re + any_re)*begins + end_re ,
                                                                Regexp::MULTI
LINE)
      result = total_re.match(@sym_code)
      # If there was no match, the syntax was wrong
      raise "SYM-ERROR: Invalid functiondef at #{get_line()},"+
                  " proper syntax is: \n\$\/function()\n\\\$" if not
result

      amount = result[0].split("\$\/").length-1
      if amount != begins
        endloop = false
        begins = amount
      end
```

```
    end while (not endloop)

    @sym_code = result.post_match()
    puts "Lexer: Matched function body. Left= #{code_left}" if
@debug_mode
    return result[0].slice(0..-3) # Slice off the end-tag
  end

  def get_code_body(parts)
    local_debug = false
    puts "@sym_code before:\n\"#{@sym_code.to_s}\"" if local_debug

    condition = code = next_part = nil

    re_condition = /(.*)/
    condition_match = re_condition.match(@sym_code)

    if (condition_match)
      condition = condition_match.captures[0]
      @sym_code = condition_match.post_match
      re_code = /(.*?)(#{parts})/m
      match_code = re_code.match(@sym_code)
      if (match_code)
        code = match_code.captures[0]
        next_part = match_code.captures[1]
        @sym_code = match_code.post_match
      end
    end

    puts "@sym_code after:\n\"#{@sym_code.to_s}\"" if local_debug
    condition = condition.strip
    return [code, condition, next_part]
  end

  def add_regexp(regexp, datatype)
    @regexp << [regexp, Proc.new {|x| Lexem.new(datatype, x)}]
  end
```

```
# Generates a list of regexp's. In pairs as follows:
# 1. Regexp, the matching pattern
# 2. A Code block(Proc) that creates the lexem object when called
def generate_regexp()
  def add_regexp(regexp, datatype)
    @regexp << [regexp, Proc.new {|x| Lexem.new(datatype, x)}]
  end

  puts "Lexer: Re-generating all regexp" if @debug_mode
  @regexp = []

  # Multiline comment
  @regexp << [/\A\#\/.*?\\\#/m, false]
  # Comment
  @regexp << [/\A\#.*$/, false]
  # Space or Tab
  @regexp << [/\A(\ |\t)+/, false]

  # If Parts
  add_regexp(/\A\?\//, IfPart)
  add_regexp(/\A\\\?\//, ElseIfPart)
  add_regexp(/\A\\\*/, ElsePart)
  add_regexp(/\A\\\?/, EndIfPart)

  # For Parts
  add_regexp(/\A\.\.\//, ForPart)
  add_regexp(/\A\\\.\./, EndForPart)

  # While Parts
  add_regexp(/\A\|\//, WhilePart)
  add_regexp(/\A\\\|/, EndWhilePart)

  # For Parts
  add_regexp(/\A\.\.\/\s+\d+\s+\d+\s+[a-z][a-zA-Z0-9_]*/, ForPart)
  add_regexp(/\A\\\.\./, EndForPart)

  # While Parts
  add_regexp(/\A\|\//, WhilePart)
```

```
    add_regexp(/\A\\\|/, EndWhilePart)

    # Break
    add_regexp(/\A\<\-\-/, BreakOperator)

    # IncrementOperator
    add_regexp(/\A[a-z][a-zA-Z0-9\_]*\+\+/, IncrementOperator)


    # Builtin datatypes
    @regexp << [/\AInt/,
            Proc.new {|x| Lexem.new( DataType, DataType.new(Int))}]
    @regexp << [/\AFixnum/,
            Proc.new {|x| Lexem.new( DataType, DataType.new(Fixnum))}]
    @regexp << [/\AFloat/,
            Proc.new {|x| Lexem.new( DataType, DataType.new(Float))}]
    @regexp << [/\AString/,
            Proc.new {|x| Lexem.new( DataType, DataType.new(String))}]
    @regexp << [/\AVoid/,
            Proc.new {|x| Lexem.new( DataType, DataType.new(Void))}]

    # Classes
    for d_type in @datatypes
      @regexp << [/\A#{d_type}\.init\(.*\)/,
             Proc.new {|x| Lexem.new(ClassInit, x)}]
      @regexp << [/\A#{d_type.to_s}/,
             Proc.new {|x| Lexem.new( DataType, DataType.new(d_type))}]
    end

    # Comma (in function-def, lists etc..)
    @regexp << [/\A\,/,
            Proc.new{|x| Lexem.new(Comma, Comma.new())}]

    # Function definition
    @regexp << [/\A\$\//,
            Proc.new {|x| Lexem.new( FunctionDef,
FunctionDef.new(false))}]
```

```
# Class definition
@regexp << [/\A\@\/[A-Z][a-z0-9\_]*/,
        Proc.new {|x| Lexem.new( ClassDef, x)}]
@regexp << [/\A\\\@/,
        Proc.new {|x| Lexem.new( ClassEnd, "")}]
# Return statement
@regexp << [/\Areturn/,
        Proc.new {|x| Lexem.new( ReturnOperator, "")}]

# Print-operator without newline
@regexp << [/\A\<\<\</,
        Proc.new {|x| Lexem.new(PrintOperator, true)}]
# Print-operator with newline
@regexp << [/\A\<\</,
        Proc.new {|x| Lexem.new(PrintOperator, false)}]

# Input-operator
@regexp << [/\A\>\>/,
        Proc.new {|x| Lexem.new(InputOperator, x)}]

# Modifier-operators
@regexp << [/\A(\+\=|\-\=|\*\=|\/\=|\%\=|\*\*\=|\^\=)/,
        Proc.new {|x| Lexem.new(ModifierOperator, x)}]
@regexp << [/\A\-\=/,
        Proc.new {|x| Lexem.new(ModifierOperator, x)}]
# Comparison-operators
@regexp << [/\A\>\=/,
        Proc.new {|x| Lexem.new(Operator, x)}]
@regexp << [/\A\<\=/,
        Proc.new {|x| Lexem.new(Operator, x)}]
@regexp << [/\A[\<\>]/,
        Proc.new {|x| Lexem.new(Operator, x)}]

# Logical operators
add_regexp(/\A\&\&/, Operator)
add_regexp(/\A\|\|/, Operator)
# Equality operator
add_regexp(/\A\=\=/, Operator)
```

```
# Negate equality operator
add_regexp(/\A\!\=/, Operator)
# Equality operator
add_regexp(/\A\=\=/, Operator)
# Negate equality operator
add_regexp(/\A\!\=/, Operator)

# Assignment
@regexp << [/\A\=/,
        Proc.new{|x| Lexem.new(AssignmentOperator, x)}]
# Float
@regexp << [/\A\d+\.\d+/,
        Proc.new {|x| Lexem.new(Float, x.to_f)} ]
# Negative Float
@regexp << [/\A-\d+\.\d+/,
        Proc.new {|x| Lexem.new(Float, x.to_f)} ]
# Integer
@regexp << [/\A\d+/,
        Proc.new {|x| Lexem.new(Integer, x.to_i)} ]
# Negative Integer
@regexp << [/\A-\d+/,
        Proc.new {|x| Lexem.new(Integer, x.to_i)} ]

# List/Dict indexing
@regexp << [/\A[a-z][a-zA-Z0-9\_]*\[.+\]/,
        Proc.new {|x| Lexem.new(Indexing, x)} ]

# List start
@regexp << [/\A\[/,
        Proc.new {|x| Lexem.new(ListStart, x)} ]
# List end
@regexp << [/\A\]/,
        Proc.new {|x| Lexem.new(ListEnd, x)} ]

# Dict relation, used to seperate each element
@regexp << [/\A\:/,
        Proc.new {|x| Lexem.new(DictRelation, x)} ]
```

```
    # Dict start
    @regexp << [/\A\{/,
            Proc.new {|x| Lexem.new(DictStart, x)} ]

    # Dict end
    @regexp << [/\A\}/,
            Proc.new {|x| Lexem.new(DictEnd, x)} ]

    # Operators
    @regexp << [/\A[\*]{2}/x,
            Proc.new {|x| Lexem.new(Operator, x)}]
    @regexp << [/\A[\+\*\/\-\^\%]/x,
            Proc.new {|x| Lexem.new(Operator, x)}]

    # Terminator (newline)
    @regexp << [/\A(\r|\n)+/,
            Proc.new{|x| Lexem.new(Terminator, "Newline")}]

    # String
    @regexp << [/\A\".*?\"/m,
            Proc.new{|x| Lexem.new(String, x)}]

    # Method call
    @regexp << [/\A[a-z][a-zA-Z0-9\_]*\.\w+\((( )*\-?\"?[0-9A-Za-z]+\.?
[0-9]*\"?)?(,( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)*( )*\)/,
            Proc.new {|x| Lexem.new( MethodCall, x)}]
    # Function call
    @regexp << [/\A\w+\((( )*\-?\"?[0-9A-Za-z]+\.?[0-9]*\"?)?(,( )*\-?\"?
[0-9A-Za-z]+\.?[0-9]*\"?)*( )*\)/,
            Proc.new {|x| Lexem.new(FunctionCall, x)}]

    # Variable name
    @regexp << [/\A[a-z][a-zA-Z0-9\_]*/,
            Proc.new{|x| Lexem.new(Variable, x)}]
    # Constant name
    @regexp << [/\A[A-Z][A-Z0-9\_]*/,
            Proc.new{|x| Lexem.new(Constant, x)}]
```

```
    # LParanthesis
    @regexp << [/\A\(/,
            Proc.new {|x| Lexem.new(LParanthesis, "(")}]
    # RParanthis
    @regexp << [/\A\)/,
            Proc.new {|x| Lexem.new(RParanthesis, ")")}]

    # Syntax error catching
    # Function end tag
    @regexp << [/\A\\\$/,
            Proc.new {|x| raise "SYM-ERROR #{get_line()}:" +
                                                " Unmatched function end
tag"}]
  end

  # Used by SymParser, returns a Lexem-object
  # Iterates over all rules generated by generate_regexp()
  # If there's a match on the regexp, the Proc is run
  def get_next_lexem()
    puts "Lexer: Getting a lexem code left='#{code_left()}'" if
@debug_mode
    if @return_previous
      puts "Lexer: Returning previous ( #{@previous_lexem.val}, "+
                        "it's a #{@previous_lexem.type} )" if @debug_mode
      @return_previous = false
      return @previous_lexem
    end

    if @sym_code.length == 0
      puts "Lexer: Out of code, returning EOF" if @debug_mode
      return Lexem.new(EOF, 0)
    end

    error = true
    # Search through all regexp's and look for a match
    for i in 0..@regexp.length-1
      match = @regexp[i][0].match(@sym_code)
```

```
      if (match)
        @sym_code = match.post_match
        @line_code += match[0]
        puts "Lexer: Match on regexp for #{@regexp[i][1].call.type if
             @regexp[i][1]}=#{@regexp[i][0]} code left=#{code_left()}" if
                                                          @debug_mode and
                                                                @regexp[i
][1]

        # If there is a block to call, call it
        if (@regexp[i][1])
          lexem = @regexp[i][1].call(match.to_s)
          error = false
        else
          # Otherwise re-call ourselves at the new position
          return self.get_next_lexem()
        end
        break
      end #end match
    end # end for

    # If there was an error, print everything from this point until next
line
    raise "SYM-ERROR: Invalid syntax, " +
                              "could not recognize: #{get_line()}\n" if
error
    @grand_previous_lexem = @previous_lexem
    @previous_lexem = lexem

    puts "Lexer: Returning #{lexem.val}, it's a #{lexem.type}" if
                                      @debug_mode and lexem.type !=
Terminator
    puts "Lexer: Code left: #{code_left()}" if @debug_mode
    puts "----" if @debug_mode
    return lexem
  end

  def return_previous()
    puts "Lexer: Return previous-flag has been set" if @debug_mode
```

```
    @return_previous = true
  end

  def return_next()
    puts "Lexer: Disabling previous flag" if @debug_mode
    @return_previous = false
  end

  def get_grand()
    return @grand_previous_lexem
  end

end
```

5.4.2 parser.rb

```
# -*- coding: utf-8 -*-
require 'Backend/lexer'
require 'Backend/scope'
class SymParser
  def initialize(debug_mode = false, silent_mode = false)
    @debug_mode = debug_mode
    @silent_mode = silent_mode    # Print does nothing if silent_mode is
true
    @lexer = nil

    # All scopes, last one is "more local"
    @scopes = []
    # Create global scope
    @scopes << Scope.new({"NULL" => nil,
                          "SYM_VERSION" => "1.0",
                          "sym_max_recursion"=>100,       # Max recursion
depth
                          "sym_current_recursion"=>0,     # Current depth
                          "true"=>true,
                          "false"=>false
                          }, @debug_mode)

    @in_fun = nil         # Reference to current function we're currently
in
    @return_level = 0     # Keeps track of scope depth
    @return_value = 0     # When a function returns, the value is saved
here
    @break = false        # True=Break out of parsing ASAP

    @break_loop = false
    @loop_level = 0                  # Keeps track of loop depth
    @profile_mode = false  # Profile code and create report
  end

  # Input is Sym code, as string
  # Also used to call functions, methods etc.
  def parse(input)
```

```
    # Save the old lexer (we need to be able to return after call)
    old_lexer = @lexer

    # Start iterating over the new code until end of file
    @lexer = SymLexer.new(input, @debug_mode)
    while evaluate_statement()
    end

    @lexer = old_lexer
  end

private
  # Searches all scopes for an object and returns its _name_
  # Argument is the object to be found, as actual reference
  def find_name(object)
    puts "Parser: Trying to find #{object}" if @debug_mode

    # Iterate from back and up
    for i in (@scopes.length-1).downto(0)
        obj = @scopes[i].get_object_name(object)
        return obj if obj != nil # If found, return it
    end
    raise "SYM-ERROR: #{name} not found"
  end

  # Searches all scopes for an object and returns its _reference_
  # type is only used as debug message
  # if raise_error=true, an error will be raised if the object isn't
found
  def seek_storage(name, type = Variable, raise_error = true)
    puts "Parser: seek_storage. Trying to find #{type} #{name}" if
@debug_mode

    # If we are in a function, use it's lexical scope instead of dynamic
scope
    if @in_fun != nil
      old_scopes = @scopes
      @scopes = @in_fun.get_scope_ref()
      @scopes << old_scopes[-1]         # Also add the function local
```

```
scope

#       for sc in @scopes
#         puts sc.to_s
#       end
#       puts "-----------------"
    end

    # Iterate from back and up
    for i in (@scopes.length-1).downto(0)
        obj = @scopes[i].get_name(name)

        @return_level = i
        # When we return the level-pointer will be invalid because the
old
        # scope is restored, add the length-difference to compensate
        @return_level += (old_scopes.length - @scopes.length) if
@in_fun != nil

        # Return to the previous scope if we are in a function
        @scopes = old_scopes if obj != nil and @in_fun != nil
        return obj if obj != nil # If found, return it
    end

    raise "SYM-ERROR #{@lexer.get_line()}: #{name} not found" if
raise_error
    return false
  end

  # Evaluates statement and runs the apppropiate action
  def evaluate_statement()
    puts "Parser: Entering evaluate_statment()" if @debug_mode

    # Break from function/method
    if @break
      @break = false
      return false
    end
```

```
    # Break from loop
    if @break_loop
      return false
    end

    @lexer.return_next()
    lexem = @lexer.get_next_lexem()

    if lexem.type == EOF
      return false
    elsif lexem.type == PrintOperator
      print_operator(lexem)
    elsif lexem.type == InputOperator
      input_operator()
    elsif lexem.type == ReturnOperator
      return_operator()
    elsif lexem.type == AssignmentOperator
      assignment_operator()
    elsif lexem.type == ClassDef
      class_def(lexem)
    elsif lexem.type == FunctionDef
      # If it was an end-def, raise error
      raise "SYM-ERROR #{@lexer.get_line()}: Unmatched function-end" if
                                                                    lexem.v
al.end
      ans = function_def()
      name = ans[0]
      ret = ans[1]
      code = ans[2]
      args = ans[3]
      scope_ref = ans[4]
      obj = Function.new(name, ret, code, args, scope_ref)
      @scopes[-1].store_object(name, obj)
    elsif lexem.type == Terminator
      # Flush "this line" buffer in lexer
      @lexer.flush_line()
      # evaluate next statement
      evaluate_statement()
```

```
    elsif lexem.type == Variable or lexem.type == Constant
      l = @lexer.get_next_lexem()
      if l.type == AssignmentOperator
        assignment_operator()
      elsif l.type == ModifierOperator
        modifier_operator(l.val)
      elsif l.type == MethodCall
        call_method(l, lexem.val)
      else
        if lexem.type == Constant and l.type == Variable
          raise "SYM-ERROR #{@lexer.get_line()}: Invalid identifier.
First letter of a variable must be lower case. Constants must be all
upper case"
        else
          raise "SYM-ERROR #{@lexer.get_line()}: Invalid statement " +
                      "#{lexem.val}"
        end


      end
    elsif lexem.type == FunctionCall
      call_function(lexem)
    elsif lexem.type == ClassInit
      class_init(lexem)
    elsif lexem.type == MethodCall
      r = /(.+)\./
      match = r.match(lexem.val)
      raise "SYM-ERROR #{@lexer.get_line()}: Invalid " +
                                        "methodcall. #{lexem.val}" if not
match
      call_method(lexem, match[1])


    elsif lexem.type == ModifierOperator
      modifier_operator(lexem.val)
    elsif lexem.type == IfPart
      conditional_statement()
    elsif lexem.type == ForPart
      iterative_for_statement()
    elsif lexem.type == WhilePart
      iterative_while_statement()
```

```
    elsif lexem.type == BreakOperator
      break_operator()
    elsif lexem.type == IncrementOperator
      increment_operator(lexem.val)
    elsif lexem.type == DictEnd
      raise "SYM-ERROR #{@lexer.get_line()}: Unmatched dictionary end, }"
    elsif lexem.type == ListEnd
      raise "SYM-ERROR #{@lexer.get_line()}: Unmatched list end, ]"
    elsif lexem.type == Comma
      raise "SYM-ERROR #{@lexer.get_line()}: Unknown comma, " +
                                              "likely from an unmatched
list"
    else
      raise "SYM-ERROR #{@lexer.get_line()}: " +
                                        "Invalid statement."
    end

    return true
  end

  def increment_operator(lexem_val)
    local_debug = false

    puts "Parser: Entering increment_operator()" if @debug_mode or
local_debug
    lexem_val = lexem_val.slice(0, lexem_val.length-2)
    found_it = @scopes[-1].get_name(lexem_val)
    begin
      Float(found_it)
    rescue # If value can't be floated, it can't be a number
      raise "SYM-ERROR: #{@lexer.get_line().strip()}: Not a number"
    else
      # Is numeric
      found_it += 1
      @scopes[-1].store_object(lexem_val, found_it)
    end
  end
```

```
 def calculate_expr()
  puts "Parser: Entering calculate_expr()" if @debug_mode
  factor1 = calculate_factor()

  lexem = @lexer.get_next_lexem()
  while lexem.val == "+" or lexem.val == "-"
    factor2 = calculate_factor()

    if lexem.val == "+"
      factor1 += factor2
    else
      factor1 -= factor2
    end

    lexem = @lexer.get_next_lexem()
  end
  @lexer.return_previous()

  return factor1
end

 def calculate_factor()
    puts "Parser: Entering calculate_factor()" if @debug_mode
    factor1 = get_number()

    lexem = @lexer.get_next_lexem()

    allowed_operators = ["*", "/", "**", "^", "%", "<", ">", "<=", ">=",
"==", "!=", "||", "&&"]
     while allowed_operators.include?(lexem.val)
      factor2 = get_number()
      #return factor2 if factor2.type == SymObject

      if lexem.val == "*"
           raise "Operation not allowed" if factor2.class == String
        factor1 *= factor2
      elsif lexem.val == "**" or lexem.val == "^"
           check_numeric_error(factor1, factor2, lexem.val)
```

```
        factor1 **= factor2
      elsif lexem.val == "/"
        raise "SYM-ERROR #{@lexer.get_line()}: Division by zero" if
factor2.ceil == 0
          check_numeric_error(factor1, factor2, lexem.val)
        factor1 /= factor2
      elsif lexem.val == "%"
          check_numeric_error(factor1, factor2, lexem.val)
          raise "SYM-ERROR #{@lexer.get_line()}: Division by zero" if
factor2.ceil == 0
        factor1 %= factor2
      elsif lexem.val == "<"
          check_numeric_error(factor1, factor2, lexem.val)
        factor1 = factor1 < factor2
      elsif lexem.val == "<="
          check_numeric_error(factor1, factor2, lexem.val)
        factor1 = factor1 <= factor2
      elsif lexem.val == ">="
          check_numeric_error(factor1, factor2, lexem.val)
        factor1 = factor1 >= factor2
      elsif lexem.val == ">"
          check_numeric_error(factor1, factor2, lexem.val)
        factor1 = factor1 > factor2
       elsif lexem.val == "=="
      factor1 = (factor1 == factor2)
       elsif lexem.val == "!="
      factor1 = (factor1 != factor2)
         elsif lexem.val == "||"
      factor1 = 1 if factor1 == true
      factor1 = 0 if factor1 == false
      factor2 = 1 if factor2 == true
      factor2 = 0 if factor2 == false
      factor1 = not(factor1 == 0 and factor2 == 0)
    elsif lexem.val == "&&"
      factor1 = factor1 and factor2
       elsif lexem.val == "=="
      factor1 = (factor1 == factor2)
       elsif lexem.val == "!="
```

```
      factor1 = (factor1 != factor2)
    end

      lexem = @lexer.get_next_lexem()
    end
    @lexer.return_previous()

    return factor1
  end

  def get_number()
    puts "Parser: Entering get_number()" if @debug_mode
    lexem = @lexer.get_next_lexem()

    if lexem.type == FunctionCall
      return call_function(lexem)
    elsif lexem.type == ListStart
      return get_list(lexem)
    elsif lexem.type == ListEnd
      return lexem
    elsif lexem.type == Indexing
      return get_by_index(lexem)
    elsif lexem.type == DictStart
      return get_dict(lexem)
    elsif lexem.type == DictEnd
      return lexem
    elsif lexem.type == ClassInit
      return class_init(lexem)

    elsif lexem.type == MethodCall
      r = /(.+)\./
      match = r.match(lexem.val)
      raise "SYM-ERROR #{@lexer.get_line()}: Invalid " +
                                    "methodcall. #{lexem.val}" if not
match
      return call_method(lexem, match[1])

    elsif lexem.type == LParanthesis
```

```
      value = calculate_expr()

      # Expected right-paranthesis
      er = @lexer.get_next_lexem()
      raise "SYM-ERROR #{@lexer.get_line()}: Unbalanced paranthesis got
#{er.type}" if er.type != RParanthesis
    elsif lexem.type == Integer or lexem.type == Float
      value = lexem.val
    elsif lexem.type == String
      value = lexem.val
    elsif lexem.type == Variable
      value = seek_storage(lexem.val)
    elsif lexem.type == Constant
      value = seek_storage(lexem.val, Constant)
    elsif lexem.type == Terminator
      return lexem
    elsif lexem.type == MethodCall
      method_call(lexem)
    else
      raise "SYM-ERROR #{@lexer.get_line()}: Not a correct number
#{lexem.val}|#{lexem.type}"
    end

    puts "Parser: Get_number returns #{value}" if @debug_mode
    return value
  end

  # Gets a certain index from a list or dict
  def get_by_index(lexem)
    r = /(?<name>[a-z][a-zA-Z0-9\_]*)\[(?<index>.+)\]/
    md = r.match(lexem.val)

    name = md["name"]
    index = md["index"]
    obj_ref = seek_storage(name)
    # The index might be an identifier, look it up if possible
    index_var = seek_storage(index, Variable, false)
```

```
    if obj_ref.class == SymList
      # Must be an integer, try to convert it if it's a string
      begin
        # If index is a variable, use it
        if index_var
          index = Integer(index_var)
        else
          index = Integer(index)
        end
      rescue Exception => e
        raise "SYM-ERROR #{@lexer.get_line} : Tried to access index "+
                                              "#{index}, but must be an
integer"
      end
      # Index was converted to integer, return the value
      raise "SYM-ERROR #{@lexer.get_line} : Arrayindex out of bounds, "+
              "tried to access index #{index}, but array is defined to
index "+
                    "#{obj_ref.list.length-1}" if index >
obj_ref.list.length-1

      return obj_ref.list[index]
    elsif obj_ref.class == SymDict
      begin
        if index_var
          e_index = index_var
        else
          # Evaluate index, it is currently only stored as string, could
be anything
          temp_lexer = SymLexer.new(index)
          e_index = temp_lexer.get_next_lexem().val
        end
      rescue Exception => e
        raise "SYM-ERROR #{@lexer.get_line} : Could not evaluate "+
                                    "index #{index} in dictionary
#{name}"
      end

      return obj_ref.get_value(e_index)
```

```
    else
      raise "SYM-ERROR #{@lexer.get_line} : Unable to find #{name}"
    end
  end

  def get_dict(lexem)
    puts "Parser: Entering get_dict()" if @debug_mode
    begins = 1
    ends = 0
    dict = {}
    # 0 = expteced key
    # 1 = expected seperator
    # 2 = expected value
    # 3 = expected comma
    expected = 0
    key = ""
    value = ""

    while true
      lex = @lexer.get_next_lexem()
      if lex.type == DictStart
        begins += 1
      elsif lex.type == DictEnd
        ends += 1
      end

      if begins == ends
        break
      end

      if lex.type == DictStart
        raise "SYM-ERROR #{@lexer.get_line()} : Dictionary expected "+
                                                "seperator" if expected
== 1
        raise "SYM-ERROR #{@lexer.get_line()} : Dictionary expected "+
                                                "comma" if expected ==
3
        obj = get_dict(lex)
```

```
        if expected == 0
          key = obj
          expected = 1

          ends += 1 # If successfully got dict, there was an end too
          @lexer.return_next()
          puts "Parser: Digested dictionary key #{key}" if @debug_mode
          next
        elsif expected == 2
          value = obj
          expected = 3
          dict[key] = value

          ends += 1 # If successfully got dict, there was an end too
          @lexer.return_next()
          puts "Parser: Digested dictionary value #{value}" if
@debug_mode
          next
        end
      elsif lex.type == Terminator
        raise "SYM-ERROR #{@lexer.get_line()} : Unmatched dictionary
tags"
      elsif lex.type == Comma
        if expected == 3
          expected = 0
        else
          raise "SYM-ERROR #{@lexer.get_line} : Unexpected comma in
dictionary"
        end
      elsif lex.type == DictRelation
        if expected == 1
          expected = 2
        else
          raise "SYM-ERROR #{@lexer.get_line()}: Unexpected dict-
seperator"
        end
      elsif expected == 0
        @lexer.return_previous()
        key = calculate_expr()
```

```
        expected = 1
        puts "Parser: Digested dictionary key #{key}" if @debug_mode
      elsif expected == 2
        @lexer.return_previous()
        value = calculate_expr()
        expected = 3
        dict[key] = value
        puts "Parser: Digested dictionary value #{value}" if @debug_mode
      else
        raise "SYM-ERROR #{@lexer.get_line} : Syntax error, invalid
dictionary"
      end
    end

    puts "Parser: Dict created as #{dict}" if @debug_mode
    @lexer.return_previous()
    obj = SymDict.new(dict)
    puts "Parser: Returning from get_dict=#{dict}" if @debug_mode
    return obj
  end

  def get_list(lexem)
    puts "Parser: Entering get_list()" if @debug_mode
    begins = 1
    ends = 0
    list = []
    # This is used to detect errors with comma's
    # true=expected an element
    # false=expected comma
    element_allowed = true

    while true
      lex = @lexer.get_next_lexem()
      if lex.type == ListStart
        begins += 1
      elsif lex.type == ListEnd
        ends += 1
      end
```

```
      if begins == ends
        break
      end

      if lex.type == ListStart
        obj = get_list(lex)
        list << obj
      elsif lex.type == Terminator
        raise "SYM-ERROR #{@lexer.get_line()} : Unmatched list tags"

      elsif lex.type == Comma
        if element_allowed
          raise "SYM-ERROR #{@lexer.get_line()} : Duplicate comma"
        else
          element_allowed = true
        end
      elsif element_allowed
        element_allowed = false
        @lexer.return_previous()
        element = calculate_expr()
        if element.class != Lexem
          puts "Parser: Adding the #{element.class}=#{element} to list"
if
                                                                    @d
ebug_mode
          list << element
        end
        element_allowed = false
      else
        raise "SYM-ERROR #{@lexer.get_line()} : Invalid list, " +
                                        "expected comma to seperate
elements"
      end
    end

    puts "Parser: List created as #{list}" if @debug_mode
    @lexer.return_previous() # Parser must find the terminator in main
loop
```

```
    obj = SymList.new(list)
    return obj
  end

  def return_operator()
    puts "Parser: Entering return_operator()" if @debug_mode

    begin
      val = calculate_expr()
    rescue Exception=>e
      raise "SYM-ERROR #{@lexer.get_line()}: Could not " +
                                  "successfully return from
#{@in_fun.name}"
    end

    raise "SYM-ERROR #{@lexer.get_line()}: Tried to return"  +
                                " outside function or method" if not
@in_fun

    # Perform typecheck
    # Void=Don't perform typecheck
    # Int is alias for fixnum, so allow them to be interchanged
      raise "SYM-ERROR #{@lexer.get_line()}: Invalid returntype, can't" +
        "  return a #{val.class}, expected to return
#{@in_fun.returntype}" if
                                          @in_fun.returntype != Void
and
                                          @in_fun.returntype !=
val.class and
                        not (@in_fun.returntype == Int and val.class ==
Fixnum)

    @return_value = val
    @break = true # Break main iteration when possible
    puts "Parser: return value set to #{@return_value }" if @debug_mode
  end

  def class_init(lexem)
    puts "Parser: Creating an object with #{lexem.val}" if @debug_mode
```

```
    # Use regexp to find each seperate part
    match = /(.*)\.(.+)\((.*)\)/.match(lexem.val)
    name = match[1]
    args = match[3]

    class_obj = seek_storage(name, Class)
    raise "SYM-ERROR #{@lexer.get_line()}: Couldn't find " +
        "init constructor in class #{name}" if not
class_obj.methods["init"]

    # Create the object
    object = SymObject.new(name, class_obj.variables, class_obj.methods)
    object.scope.store_object("self", object)

    # Add object scope
    object_scope = object.scope
    @scopes << object_scope

    # Re-use the call function code, but it expects a lexem object
    # So we need to trick it
    l = Lexem.new(FunctionCall, "init"+"("+args+")")
    call_function(l)

    @scopes.delete(object_scope)
    return object
  end

  def call_method(lexem, objname)
    puts "Parser: Entering call_method()" if @debug_mode

    # Look up the variables used for recursion
    max_rec = seek_storage("sym_max_recursion", Variable, false)
    c_rec = seek_storage("sym_current_recursion", Variable, false)
    return @return_value if c_rec >= max_rec and not max_rec == -1

    # Use regexp to get each part
    match = /\.(.+?)\((.*)\)/.match(lexem.val)
    name = match[1]
```

```
    args = match[2]

    obj_ref = seek_storage(objname, SymObject)

    method_ref = obj_ref.methods[name]
    raise "SYM-ERROR #{@lexer.get_line()}: Method #{name} could not " +
                          "be found in object #{objname}" if not
method_ref

    # Object scope
    objscope = obj_ref.scope
    @scopes << objscope
    @scopes[-1].store_object("self", obj_ref)

    # Set current function to self, restore old reference afterwards
    old_ref = @in_fun
    @in_fun = method_ref

    # Run the method
    puts "Parser: Calling method #{name} in object #{objname}" if
@debug_mode
    # Re-use the code in call_function, it expects a lexem object
    l = Lexem.new(FunctionCall, name+"(#{args})")
    call_function(l)

    @scopes.delete(objscope)
    return @return_value
  end

  # NOTE: Cluttered code, this is due to the fact that
  # it's re-used in method calls and class init
  def call_function(lexem)
    puts "Parser: Functioncall with #{lexem.val}" if @debug_mode

    # Make sure we havn't went above the maximum recursion depth
    # -1 means don't have a limit
    max_rec = seek_storage("sym_max_recursion", Variable, false)
    c_rec = seek_storage("sym_current_recursion", Variable, false)
```

```
    return @return_value if c_rec >= max_rec and not max_rec == -1

    re = /(.+)\((.*)\)/
    name = lexem.val.split(re)[1]
    args = lexem.val.split(re)[2]   # A string of all arguments
    argument_list = nil
    argument_list = args.split(",") if args # An array of all arguments

    fun_ref = seek_storage(name, Function) # Function reference
    raise "SYM-ERROR #{@lexer.get_line()}: Function #{name} is not
defined" if
                                                                  not
fun_ref
    arguments = {}
    argval = nil
    index = 0

    for argname in fun_ref.arguments.keys

      begin
        # First look for objects of this name, if that doesn't work
        # Evaluate it as a literal
        argval = seek_storage(argument_list[index], Variable, false)
        argval = eval(argument_list[index].to_s) if not argval
      rescue Exception => e
        raise "SYM-ERROR #{@lexer.get_line()}: Invalid functioncall." +
                    "Could not evaluate argument
#{argument_list[index].to_s}"
      end
      raise "SYM-ERROR #{@lexer.get_line()}: Invalid functioncall." +
                               " Argument #{argname} not supplied or " +
                                    "it was something invalid" if not
argval

      # Raise error if it's not the right kind of variable
      # Don't perform typecheck if it's of void type
      # Int is just an alias for fixnum, so allow either one
      if fun_ref.arguments[argname] != Void and
          fun_ref.arguments[argname] != argval.class and
```

```
            not (fun_ref.arguments[argname] == Int and argval.class ==
Fixnum)

        # If Int is expected the variable might be a Fixnum
        # Because that's how integers are stored internally in ruby
        raise "SYM-ERROR #{@lexer.get_line()}: Invalid functioncall."+
          " Argument #{argname} expected type
#{fun_ref.arguments[argname] if
            fun_ref.arguments[argname] != Fixnum}" +
            "#{Int if fun_ref.arguments[argname] == Fixnum} but "+
            "got #{argval.class if argval.class != Fixnum}"+
            "#{Int if argval.class == Fixnum}"
      end
      arguments[argname] = argval
      index += 1
    end

    # Create a new local scope for the function, and put the arguments in
it
    newscope = Scope.new(arguments, @debug_mode)
    @scopes << newscope

    puts "Parser: Calling function #{name}" if @debug_mode

    # Set current function to self, restore old reference afterwards
    old_ref = @in_fun
    @in_fun = fun_ref
    @scopes[0].storage["sym_current_recursion"] += 1

    self.parse(fun_ref.code) # Call the function code
    # Reset
    @in_fun = old_ref
    @scopes.delete(newscope)
    @scopes[0].storage["sym_current_recursion"] -= 1
    return @return_value
  end

  def class_def(lexem)
    puts "Parser: Entering class_def()" if @debug_mode
```

```
    # Terminators(newlines) are allowed in infinite amounts in a class
def
    # This function is used to removes them
    def get_lex()
      lex = @lexer.get_next_lexem()
      if lex.type != Terminator
        return lex
      else
        return get_lex()
      end
    end

    # Class name
    name_lex = lexem
    raise "SYM-ERROR #{@lexer.get-line}: " if name_lex.type == Variable
    name = name_lex.val.slice(2..-1)

    # Class variables
    variables = {}
    next_lex = get_lex()
    not_allowed = ["return", "SYM_VERSION", "sym_max_recursion",
                       "sym_current_recursion", "true", "false"]
    while (next_lex.type != FunctionDef and next_lex.type != ClassEnd)
      raise "SYM-ERROR #{@lexer.get_line}: Can not create reserved "+
             "identifier #{next_lex.val}" if not_allowed.include?
(next_lex.val)
      raise "SYM-ERROR #{@lexer.get_line()}: Can not "+
        "create reserved identifier return " if next_lex.type ==
ReturnOperator

      variables[next_lex.val] = 0
      next_lex = get_lex()
    end




    puts "Parser: Attempting to read class methods list" if @debug_mode
```

```
    # Methods
    methods = {}
    while (next_lex.type == FunctionDef)
      ans = function_def()
      # name, ret, code, args, scope_ref
      obj = Function.new(ans[0], ans[1], ans[2], ans[3], ans[4])
      methods[obj.name] = obj
      next_lex = get_lex()
    end

    if @debug_mode
      puts "Parser: Creating a class:"
      puts "\tName: #{name}"

      # Class variables
      if variables.length > 0
        puts "\tVariables: "
        for var in variables.keys
          puts "\t\t#{var}"
        end
      end

      # Methods, their returntype and possible arguments
      if methods.length > 0
        puts "\tMethods: "
        for fun_name in methods.keys
          puts "\t\t#{fun_name}:"
          puts "\t\t\tReturns: #{methods[fun_name].returntype} "
          if methods[fun_name].arguments.length > 0
            puts "\t\t\tArgs:"
            for args in methods[fun_name].arguments.keys
              puts "\t\t\t\t#{ methods[fun_name].arguments[args]},
#{args}"
            end
          end
        end
      end
```

```
    end

    # Create the class and save it
    c_obj = Class.new(name, variables, methods)
    @scopes[-1].store_object(name, c_obj)
    @lexer.add_datatype(name)
  end

  # Creates a function and stores it in scope.
  # Gets different lexems from lexer depending on arguments.
  # If there are no arguments, it will get "FunctionCall" lexem, etc...
  def function_def()
    puts "Parser: Entering function_def()" if @debug_mode
    # Terminators(newlines) are allowed in infinite amounts in a function
def
    # This function is used to removes them
    def get_lex()
      lex = @lexer.get_next_lexem()
      if lex.type != Terminator
        return lex
      else
        return get_lex()
      end
    end

    # Returntype:
    ret_lex = get_lex()
    name_lex = ""
    # If there was no returntype specified, this lex is the name instead
    # We also default to type Void
    if ret_lex.type != DataType
      name_lex = ret_lex
      ret_lex = Lexem.new(DataType, DataType.new(Void))
    else
      name_lex = get_lex()
    end
    name = name_lex.val
```

```
    # Argument hashes
    # Key=name, as string
    # Value=type(Int, String etc..), Void=Don't force anything
    args = {}
    # Recognised as functioncall if there's no arguments
    if name_lex.type == FunctionCall
      reg = /\A(.+)\((.*)\)/
      md = reg.match(name_lex.val)

      # Raise error if there was no match
      if md
        name = md[1]

        # If there was some (none type specified) arguments, save them
(as void)
        if md[2]
          for f_arg in md[2].split(",")
            args[f_arg] = Void
          end
        end

      else
        raise "SYM-ERROR #{@lexer.get_line()}: Invalid functiondef"
      end
    else
      # L paranthesis:
      lp_lex = get_lex()

      # Arguments:
      previous = lp_lex
      current_lex = nil

      while (true) # Raise's implement syntax check
        current_lex = get_lex()

        if current_lex.type == Variable
          if previous.val.class == DataType
            args[current_lex.val] = previous.val.datatype
```

```
          else
            args[current_lex.val] = Void
          end

          previous = current_lex
        elsif current_lex.type == DataType
          previous = current_lex
        elsif current_lex.type == Comma
          raise "SYM-ERROR #{@lexer.get_line()}: Invalid functiondef.
Proper syntax is: \n$/Returntype name(Arguments(Datatype Name)*)\n
Body\n\\$\n * =Arguments are optional, leave as () for no arguments" if
previous.type != Variable
        elsif current_lex.type == RParanthesis # We reached the end of
the definition
          # Raise an error if the last lexem was a datatype, it needs to
have a name
          raise "SYM-ERROR #{@lexer.get_line()}: Invalid functiondef.
Proper syntax is: \n$/Returntype name(Arguments(Datatype Name)*)\n
Body\n\\$\n * =Arguments are optional, leave as () for no arguments" if
previous.type != Variable and previous.type != LParanthesis
          break
        else
          raise "SYM-ERROR #{@lexer.get_line()}: Invalid functiondef.
Proper syntax is: \n$/Returntype name(Arguments(Datatype Name)*)\n
Body\n\\$\n * =Arguments are optional, leave as () for no arguments"
        end
    end
    end

    # Perform check for reserved words in function arguments
    for a in args.keys
      not_allowed = ["return", "SYM_VERSION", "sym_max_recursion",
                                  "sym_current_recursion", "true",
"false"]

      raise "SYM-ERROR : Can not use reserved identifier #{a} as
argument" if
                                                not_allowed.include?(a)
    end



    # Code
```

```
    code = @lexer.get_function_body()
    ret = ret_lex.val.datatype
    scope_ref = get_fun_scope()

    if @debug_mode
      puts "Parser: Creating function, using:"
      puts "  Returntype: #{ret}"
      puts "  Name: #{name}"

      puts "  Arguments: "
      for argname in args.keys
        print "    #{argname}="
        puts args[argname]
      end

      puts "  Code: '#{@lexer.code_left(code)}'"
    end

    return [name, ret, code, args, scope_ref]
  end

  def get_fun_scope()
    result = []
    for scope in @scopes
      result << scope
    end
    return result
  end

  def input_operator()
    local_debug = false
    puts "Parser: Entering input_operator()" if @debug_mode or
local_debug
    lexem = @lexer.get_next_lexem()

    if lexem.type == Variable
      val = STDIN.gets().slice(0..-2)
       @scopes[-1].store_object(lexem.val, val)
```

```
    elsif lexem.type == Constant
      val = STDIN.gets().slice(0..-2)
       constant_exists = @scopes[-1].get_name(lexem.val)
       if (constant_exists)
        raise "SYM-ERROR: Constant #{lexem.val} already in use"
       else
         @scopes[-1].store_object(lexem.val, val, true)
       end
    else
      raise "SYM-ERROR #{@lexer.get_line()}: Invalid lexem" +
                        " , expected variable/constant '#{lexem.val}'"
    end

    puts "Parser: Assigning the #{lexem.type}" +
                       " #{lexem.val} to #{val}" if @debug_mode or
local_debug
  end

  def modifier_operator(lexem_val)
    puts "Parser: Entering modifier_operator()" if @debug_mode
    puts "Parser: lexem_val: " + lexem_val.to_s if @debug_mode
    varname = @lexer.get_grand()

    if varname.type == Variable
      temp_var = seek_storage(varname.val)
      val = calculate_expr()

       check_numeric_error(temp_var, val, lexem_val) if lexem_val !=
"*="

      if (lexem_val == "+=")
        plus_val = temp_var.to_i + val
      elsif (lexem_val == "-=")
        plus_val = temp_var - val
      elsif (lexem_val == "*=")
           raise "SYM-ERROR #{@lexer.get_line()}: Invalid operation. " +
                " Right hand operand must be numeric" if !is_numeric?
(val)
        plus_val = temp_var * val
```

```
      elsif (lexem_val == "/=")
           raise "SYM-ERROR #{@lexer.get_line()}: Can't divide variable
with zero" if val.ceil == 0
        plus_val = temp_var / val
      elsif (lexem_val == "%=")
        plus_val = temp_var % val
      elsif (lexem_val == "**=" || lexem_val == "^=")
        plus_val = temp_var ** val
      else
        raise "SYM-PARSE-ERROR OCCURRED: Contact Sym Team. Report error
code #1"
      end

       @scopes[-1].store_object(varname.val, plus_val)
    elsif varname.type == Constant
      raise "Error: Constant #{varname.val.to_s} can not be respecified"
    else
      raise "Error: Invalid lexem for assignment #{varname.type}"
    end
  end

  def assignment_operator()
    puts "Parser: Entering assignment_operator()" if @debug_mode
    varname = @lexer.get_grand()
    val = calculate_expr()

    # Check if it already exists in some scope
    # If so, assign the value to this variable instead of creating a new
one
    answ = seek_storage(varname.val, varname.type, false)
    if answ
      @scopes[@return_level].store_object(varname.val, val, varname.type
== Constant)
    else # Doesn't exist already, just put it in the local scope
      # Also make sure the name is valid variable/constant
      if varname.type == Variable or varname.type == Constant
        @scopes[-1].store_object(varname.val, val)
      else
        raise "SYM-ERROR #{@lexer.get_line()}: Invalid lexem for"+
```

```
                                                  " assignment
#{@varname.type}"
      end
    end
  end

  def print_operator(lexem = nil)
    puts "Parser: Entering print_operator()" if @debug_mode
    value = calculate_expr()

    # Objects are treated specially
    if value.class == SymObject
      if value.methods.include?("__print__")
        # The object has the special __print__ method, call it
        objname = find_name(value)
        str = "#{objname}.__print__()"
        self.parse(str)
      else
        raise "SYM-ERROR #{@lexer.get_line()}: Tried to print"+
                                       " object. Define the __print__()
method."
      end
    elsif value.class == SymList or value.class == SymDict
      value.print_self()

    else # Everything not an object
      # Replace escapes in strings
      if (value.class == String)
        value = value.gsub(/\\n/, "\n")
        value = value.gsub(/\\t/, "\t")

        value = value.gsub(/\\r/, "\r")
        value = value.gsub(/\\s/, "\s")
        value = value.gsub(/\\\\/, "\\") # Double backspace
      end

      # Don't print anything if silent mode is enabled
      # Highlight output in debug mode
```

```
      if lexem and lexem.val
        print "--SYM-->" if @debug_mode and not @silent_mode
        print value if not @silent_mode
        print "<-SYM---" if @debug_mode and not @silent_mode
      else
        puts "--SYM-->" if @debug_mode and not @silent_mode
        puts value if not @silent_mode
        puts "<-SYM---" if @debug_mode and not @silent_mode
      end

      end
  end

  def is_numeric?(arg)
    return arg.is_a? Numeric
  end

  def check_numeric_error(factor1, factor2, lexem_val)
    def has_all_numeric(*args)
       for arg in args
          return false if not is_numeric?(arg)
       end
       return true
     end

     raise "SYM-ERROR: Operation #{lexem_val} invalid on
'#{factor1.to_s}' and '#{factor2.to_s}'" if not has_all_numeric(factor1,
factor2)
  end
  def conditional_statement()
    local_debug = false
    puts "##############################" if local_debug or @debug_mode
    puts "Entering conditional_statement" if local_debug or @debug_mode

    if_code = ""
     if_condition = ""
    else_if_code = []
    else_code = ""
```

```
    accumulated_code = ""

    save_to_if_code = "saving code to if code"
    save_to_else_if_code_first = "saving code to else if code (first)"
    save_to_else_if_code_second = "saving code to else if code (second)"
    save_to_else_code = "saving code to else code"

    save_to_code = save_to_if_code
    saved_to_last = save_to_code

    # Used to check multiple appearances of "else" in a flat if
structure,
    # which is strictly unallowed
    else_code_saved = false
    identity = nil

    if_level = 1

    while (if_level != 0)
      puts "if_level: #{if_level.to_s}" if local_debug or @debug_mode
      code, condition, next_part =
                @lexer.get_code_body(/(\\\?\/)|(\\\?)|(\?\/)|(\\\*)/)
      puts "ret code\n\"#{code.to_s}\"" if local_debug or @debug_mode
      puts "ret condition\n\"#{condition.to_s}\"" if local_debug or
@debug_mode
      puts "ret next_part\n\"#{next_part.to_s}\"" if local_debug or
@debug_mode

      # Keep track where code was saved last time.
      # If it has changed set code to accumulated_code
      if (saved_to_last != save_to_code)
        puts "changing saving. Now #{save_to_code}" if local_debug or
@debug_mode
        puts "saved_to_last: #{saved_to_last}" if local_debug or
@debug_mode
        if (saved_to_last == save_to_if_code)
          if_code = [accumulated_code, identity]
          puts "saved if code: #{if_code.to_s}" if local_debug or
@debug_mode
        elsif (saved_to_last == save_to_else_code)
```

```
          else_code = [accumulated_code, identity]
          puts "saved else_code: #{else_code.to_s}" if local_debug or
@debug_mode
        elsif (saved_to_last == save_to_else_if_code_first or
              saved_to_last == save_to_else_if_code_second)
          else_if_code << [accumulated_code, identity]
        end

        # Empty accumulated_code so it can be reused
        accumulated_code = ""
       else
          if_condition = condition
      end

      if (if_level == 1)
          newscope = Scope.new({"identity" => "0", }, @debug_mode)
          @scopes << newscope
        self.parse("identity = " + condition) if condition != ""
          identity = @scopes[-1].get_name("identity")
          @scopes.delete(newscope)
        puts "condition was: #{identity.to_s}" if local_debug or
@debug_mode
      end

      saved_to_last = save_to_code

      # If level of code is greater than 1 add IfParts that would
otherwise
      # disappear when getting code from get_code_body()
      if (if_level > 1)
        if (next_part == "?/" or next_part == "\\?") # IfPart or
EndIfPart
          accumulated_code += "?/ " if condition != ""
          accumulated_code += condition
        elsif (next_part == "\\*") # ElsePart
          accumulated_code += "?/ " if condition != ""
          accumulated_code += condition
        elsif (next_part == "\\?/") # ElseIfPart
          accumulated_code += "?/ " if condition != ""
```

```
          accumulated_code += condition
        end
      end

        accumulated_code += code if code != nil

        # Increase or decrease "if level" if necessary. Also add IfParts
to
        # the accumulated code, to make the code correct
      if (next_part == "?/") # IfPart
       puts "Next part was IfPart" if local_debug or @debug_mode
       if_level += 1
    elsif (next_part == "\\?") # EndIfPart
      puts "Next part was EndIfPart" if local_debug or @debug_mode
      accumulated_code += "\\?" if if_level != 1
      if_level -= 1
    elsif (next_part == "\\*") # ElsePart
      puts "Next part was ElsePart" if local_debug or @debug_mode
      accumulated_code += "\\*" if if_level != 1
    elsif (next_part == "\\?/") # ElseIfPart
      puts "Next part was ElseIfPart" if local_debug or @debug_mode
      accumulated_code += "\\" if if_level != 1
    else
      raise "SYM-ERROR #{@lexer.get_line()} #{condition.to_s}: Undefined
if structure"
      end

      # If the if structure is flat (i e the if level is == 1)
    # change save_to_code to else or else if.
        if (if_level == 1)
            if (next_part == "\\*")
              raise "SYM-ERROR ?\/ #{if_condition.to_s}: Multiple
appearances of else part" if else_code_saved
          puts "saving code to else..." if local_debug or @debug_mode
          save_to_code = save_to_else_code
          else_code_saved = true
        elsif (next_part == "\\?/")
          # Shift between variables to make sure that code is saved
correctly
```

```
          save_to_code = save_to_else_if_code_first ==
          save_to_code ? save_to_else_if_code_second :
save_to_else_if_code_first
        end
        end

        puts "acc code: #{accumulated_code.to_s}" if local_debug or
@debug_mode
  end

      if (saved_to_last == save_to_if_code)
        if_code = [accumulated_code, identity]
        puts "saved if code: #{if_code.to_s}" if local_debug or
@debug_mode
      elsif (saved_to_last == save_to_else_code)
        else_code = [accumulated_code, identity]
        puts "saved else_code: #{else_code.to_s}" if local_debug or
@debug_mode
      elsif (saved_to_last == save_to_else_if_code_first or
       saved_to_last == save_to_else_if_code_second)
        else_if_code << [accumulated_code, identity]
      end

      # Print all different codes
      puts "##############################" if local_debug or @debug_mode
      puts "if_code:\"\n#{if_code[0].to_s}\"" if local_debug or
@debug_mode
      puts "if_code condition: #{if_code[1].to_s}\"" if local_debug or
@debug_mode

      else_if_code.each_index do |i|
        puts "elseif_code:\n\"#{else_if_code[i][0]}\"" if local_debug or
       @debug_mode
        puts "elseif_condition: #{else_if_code[i][1]}" if local_debug or
       @debug_mode
      end
      puts "else_code:\"\n#{else_code[0].to_s}\"" if local_debug or
@debug_mode
      puts "------------------------------" if local_debug or @debug_mode
```

```
      code_to_run = nil
      # Choose the code to run according to each if-parts identity values
      if (if_code[1] == true)
        code_to_run = if_code[0]
      else
        # step through each else if code and see if its condition is true
        else_if_code.each_index do |i|
           if (else_if_code[i][1] == true)
             code_to_run = else_if_code[i][0]
             break
           end
        end
        # if there were no true conditions in "if" or "else if", set
code_to_run
        # to "else" code, if there are any
        if (code_to_run == nil)
           code_to_run = else_code[0] if else_code[0] != nil
        end
      end
      code_to_run = code_to_run.strip if code_to_run
      puts "final code:\n\"#{code_to_run.to_s}\"" if local_debug or
@debug_mode
      self.parse(code_to_run) if code_to_run != "" and code_to_run != nil
  end


 def iterative_for_statement()
      local_debug = false


      puts "Entering iterative_for_statement" if local_debug or
@debug_mode


      accumulated_code = ""
      condition_saved = false
      saved_condition = ""


      iterative_for_level = 1
      while (iterative_for_level != 0)
           code, condition, next_part = @lexer.get_code_body(/(\.\.\/)|
(\\\.\.)/)
```

```
          puts "returned code\n\"#{code.to_s}\"" if local_debug or
@debug_mode
          puts "ret condition\n\t\"#{condition.to_s}\"" if local_debug
or @debug_mode
          puts "ret next_part\n\t\"#{next_part.to_s}\"" if local_debug
or @debug_mode


          # static check - only check once
          if not condition_saved
              valid_num = /[1-9][0-9]*|0/


               minimal_parts = /\A(#{valid_num})\s+
(#{valid_num})\s*$/.match(condition)
               maximal_parts = nil
               if minimal_parts
                    raise "SYM-ERROR ../ #{condition.to_s}: Invalid
parts of for-loop" if not minimal_parts
                    min_val = minimal_parts.captures[0].to_i
                    max_val = minimal_parts.captures[1].to_i
                    local_val = nil
                    step_val = 0
               else
                    maximal_parts = /\A(#{valid_num})\s+
(#{valid_num})\s+([a-z][a-zA-Z0-9_]*)(\s+(([1-9][0-9]*$)|
0))?/.match(condition)
                    if maximal_parts
                         min_val = maximal_parts.captures[0].to_i
                         max_val = maximal_parts.captures[1].to_i
                         local_val = maximal_parts.captures[2].to_s
                         step_val = maximal_parts.captures[3]
                    end
                    raise "SYM-ERROR ../ #{condition.to_s}: Invalid
parts of for-loop" if not maximal_parts
               end


               if not step_val
                 match_invalid_step_val = /\A\d+\s+\d+\s+[a-z][a-zA-Z0-
9_]*\s(\s*.*)?/.match(condition)
                 if match_invalid_step_val
                    invalid_step_val =
match_invalid_step_val.captures[0].to_s
```

```
                    raise "SYM-ERROR #{@lexer.get_line().strip()}
#{condition.to_s}: " +
                    "Invalid step value --> #{invalid_step_val.to_s}
<--"
                  end
                end
                step_val = step_val.to_i

                puts "min_val: #{min_val.to_s}" if local_debug or
@debug_mode
                puts "max_val: #{max_val.to_s}" if local_debug or
@debug_mode
                puts "local_val: #{local_val.to_s}" if local_debug or
@debug_mode
                puts "step_val: #{step_val.to_s}" if local_debug or
@debug_mode
                raise "SYM-ERROR #{@lexer.get_line()} #{condition.to_s}:
Illegal values to" +
                    " for loop. Min val has to be lower than or equal
to Max val" if
                    min_val.to_i > max_val.to_i
                condition_saved = true
            end

            if (iterative_for_level > 1)
              if (next_part == "../" or next_part == "\\..")
                accumulated_code += "../" if condition != ""
                accumulated_code += condition
              end
            end

            accumulated_code += code if code != nil

            if (next_part == "../") # ForPart
              puts "Next part was ForPart" if local_debug or @debug_mode
              iterative_for_level += 1
            elsif (next_part == "\\..") # EndForPart
              puts "Next part was EndForPart!" if local_debug or
@debug_mode
              accumulated_code += "\\.." if iterative_for_level != 1
              iterative_for_level -= 1
```

```
          else
            raise "SYM-ERROR ../ #{condition}: Unfinished for structure.
\\.. missing"
          end


     end


     puts "Code at end:\"#{accumulated_code.to_s}\"" if local_debug or
@debug_mode
     step_counter = 0
     @loop_level += 1
     min_val.upto(max_val) do |i|
          @scopes[-1].store_object(local_val, i) if local_val != nil
          break if step_val >= max_val
          if step_counter == 0
            self.parse(accumulated_code)
            break if @break_loop
            step_counter = step_val+1
          end
          step_counter -= 1 if step_counter > 0
     end
     @loop_level -= 1
     @break_loop = false
     @scopes[-1].unstore_object(local_val) if local_val != nil
     puts "end of iterative_for_statement" if local_debug or @debug_mode
  end


   def iterative_while_statement()
     local_debug = false
     iterative_while_level = 1
     puts "               Entering iterative_while_statement" if
local_debug or @debug_mode
     puts "               while_level is : #{iterative_while_level}" if
local_debug


     accumulated_code = ""
     condition_saved = false
     saved_condition = ""
     while (iterative_while_level != 0)
```

```
        code, condition, next_part = @lexer.get_code_body(/(\|\/)|
(\\\|)/)
        puts "ret code\n\"#{code.to_s}\"" if local_debug or @debug_mode
        puts "ret condition\n    \"#{condition.to_s}\"" if local_debug or
@debug_mode
        puts "ret next_part\n    \"#{next_part.to_s}\"" if local_debug or
@debug_mode

        if not condition_saved
           saved_condition = condition
           condition_saved = true
           raise "SYM-ERROR #{@lexer.get_line()}: Invalid while
condition" if saved_condition == ""
        end

        if (iterative_while_level > 1)
           if (next_part == "|/" or next_part == "\\|")
             accumulated_code += "|/" if condition != ""
             accumulated_code += condition
           end
        end

        accumulated_code += code if code != nil

        if (next_part == "|/") # While
           puts "Next part was WhilePart" if local_debug or @debug_mode
           iterative_while_level += 1
        elsif (next_part == "\\|") # EndWhile
           puts "Next part was EndWhilePart" if local_debug or
@debug_mode
           accumulated_code += "\\|" if iterative_while_level != 1
           iterative_while_level -= 1
        else
           raise "SYM-ERROR #{@lexer.get_line()} #{condition}:" +
           " Unfinished while structure. \\| missing"
        end

     end
```

```
      puts "Acc code at end:\"#{accumulated_code.to_s}\"" if local_debug
or
        @debug_mode
      puts "Saved condition: \"#{saved_condition.to_s}\"" if local_debug
or
        @debug_mode
      accumulated_code = accumulated_code.strip

      @loop_level += 1
      while true
        newscope = Scope.new({"identity" => "0", }, @debug_mode)
        @scopes << newscope
        self.parse("identity = " + saved_condition)
        identity = @scopes[-1].get_name("identity")
        @scopes.delete(newscope)
        if identity
           self.parse(accumulated_code) if accumulated_code != "" and not
@break_loop
        end
        break if not identity or @break_loop
      end
      @break_loop = false
      puts "Ending iterative_while_statement" if local_debug or
@debug_mode
      @loop_level -= 1
end
end
```

5.4.3 datatypes.rb

```
# This file contains a list of all internal representations for Sym and
more

class ListStart
end

class ListEnd
end

class Indexing
end

class DictStart
end

class DictEnd
end

# Seperator
class DictRelation
end

class Operator
end

class UniOperator
end

class Expression
end

class LParanthesis
end

class RParanthesis
end
```

```
class Terminator
end

class PrintOperator
end

class InputOperator
end

class AssignmentOperator
end

class EOF
end

class Variable
end

class Constant
end

class BreakOperator
end

class IncrementOperator
end


class ComparisonOperator
end

class ModifierOperator
end

class ListContainer
end
```

```
class HashContainer
end

class Conditional
end

class IfPart
end

class ElseIfPart
end

class ElsePart
end

class EndIfPart
end

class WhilePart
end

class EndWhilePart
end

class ForPart
end

class EndForPart
end

class FunctionCall
end

class Comma # ","
end

class ClassDef
end
```

```
class ClassEnd
end

class ClassName
end

class MethodCall
end

class ClassInit
end

class ReturnOperator
end

# Aliases :
class Int < Integer
end

class Double < Float
end
class Void
end

class Regexp
  def +(re)
    Regexp.new(self.to_s + re.to_s)
  end

  def *(arg)
    Regexp.new(self.to_s * arg)
  end
end

class Lexem
  attr_accessor :type, :val
```

```
  def initialize(type = nil, val = nil)
    @type = type
    @val = val

    if type == String
      @val = @val.slice(1..-2) if @val != nil
    end

  end
end

class SymDict
  attr_reader :dict
  def initialize(dict = {})
    @dict = dict
  end

  def print_self()
    if @dict.length == 0
      print "{}"
      return
    end

    print "{"
    index = 0
    for key in @dict.keys
      index += 1
      value = @dict[key]

      if key.class == SymDict or key.class == SymList
        key.print_self()
      else
        print key
      end
      print ":"

      if value.class == SymDict or value.class == SymList
        value.print_self()
```

```
      else
        print value
      end
      print ", " if index < @dict.length
    end
    print "}"
  end

  def get_value(key)
    raise "\nSYM-ERROR: Could not evaluate dictionary statement" if
                                                          not
@dict.keys.include?(key)
    return @dict[key]
  end
end

class SymList
  attr_reader :list
  def initialize(list = [])
    @list = list
  end

  def print_self()
    if @list.length == 0
      print "[]"
      return
    end

    print "["
    for i in 0..@list.length
      element = @list[i]

      if element.class == SymList or element.class == SymDict
        element.print_self()
      else
        print element
      end
      print ", " if i < (@list.length-1)
```

```
    end
    print "]"
  end

end

class SymObject
  attr_reader :name, :variables, :methods
  attr_accessor :scope
  def initialize(name, vars, methods)
    @name = name        # Name of _class_, not the object
    @variables = vars
    @methods = methods

    s = {}
    for var_k in vars.keys
      s[var_k] = vars[var_k]
    end
    for met_k in methods.keys
      s[met_k] = methods[met_k]
    end

    @scope = Scope.new(s)
  end
end

class Class
  attr_reader :name, :variables, :methods
  def initialize(name, vars, methods)
    @name = name
    @variables = vars
    @methods = methods
  end
end

# Temporary function-definition for each lexem, the actual tag
class FunctionDef
  attr_reader :end, :code
```

```
  def initialize(end_def = false)
    @end = end_def # true if it's the end-tag
  end
end

# Internal function class, this is the actual sym-functions
class Function
  attr_reader :name, :returntype, :code, :arguments, :scope_ref
  def initialize(name, returntype, code, args, arg_scope)
    @name = name
    @returntype = returntype
    @code = code
    @arguments = args
    @scope_ref = arg_scope
  end

  def get_scope_ref()
    return @scope_ref
  end
end

# Generic type
class DataType
  attr_reader :datatype
  def initialize(type)
    @datatype = type
  end
end
```

5.4.4 scope.rb

```
#-*- coding:utf-8 -*-
require 'Backend/datatypes'

class Scope
  attr_accessor :storage

  def initialize(objects = {}, debug_mode = false)
    # Key: Name, as string
    # Value: Object reference
    @storage = objects
    @debug_mode = debug_mode
  end

  # Returns the object, from the name
  def get_name(name)
    return @storage[name]
  end

  # Returns the name, from the object
  def get_object_name(object)
    for key in @storage.keys
      return key if @storage[key] == object
    end

    return false
  end

  # Removes an object from storage
  def unstore_object(name, no_duplicate = false)
    raise "SYM-ERROR: Constant #{name} can't be deleted" if no_duplicate
and
                    @storage.include?(name)

    puts "Scope: Deleting object #{name}" if @debug_mode
    @storage.delete(name)
  end
```

```
  # Arg 1: Name of object, as string
  # Arg 2: Object to be stored
  # Arg 3: true=raise an error if there is already something by that name
  def store_object(name, object, no_duplicate = false)
    raise "SYM-ERROR: Constant #{name} already in use" if no_duplicate
and
                                                        @storage.include?
(name)

    # Some names are reserved, these are not allowed to be stored
    not_allowed = ["return", "SYM_VERSION",
                                    "sym_current_recursion", "true",
"false"]

    raise "SYM-ERROR : Can not create reserved identifier #{name}" if
                                              not_allowed.include?(name)

    puts "Scope: Storing object #{name}" if @debug_mode
    @storage[name] = object
  end

  # Get's the scope as a string, for debugging purposes
  def to_s
    res = "Scope trace:\n"
    for s_key in @storage.keys
      res += "\t#{s_key}=>#{@storage[s_key]}\n"
    end
    return res
  end
end
```