<3

- Ett älskvärt interpreterat imperativt datorspråk

We <3 you!

Hannah Börjesson och **Per Jonsson**
hanbo174@student.liu.se perjo927@student.liu.se
VT 2013. 2013-05-13.

Linköpings universitet
Innovativ Programmering åk 1
TDP019 – Projekt: Datorspråk

Omslagsbild: *Bright Red Heart-Shaped Diamond.*
Upphovsman: FreeGreatPicture.com (2010). Fri resurs.
Källa: http://www.freegreatpicture.com/other/bright-red-heart-shaped-diamond-17944
Hämtad: 2013-05-10.

Sammanfattning

I kursen ”Projekt: datorspråk” (TDP019) på utbildningsprogrammet Innovativ Programmering på Linköpings universitet, har författarna konstruerat programmeringsspråket ”<3” (som uttalas ”mindre än tre”). <3 är en symbol för ett hjärta och tanken är att det ska vara ett älskvärt språk. Syftet har varit att skapa ett språk som kan appellera till programmeringsnybörjare, med en enkel och tydlig struktur som påminner om XML samt har influenser av interpreterade och imperativa programmeringsspråk såsom Java.

Utifrån föreläsningar i kurserna TDP007 och TDP019 har vi lärt oss hantera programmeringsspråket Ruby och erhållit teoretiska grunder i programspråkskonstruktion. Metoden har varit dels av karaktären *trial-and-error*, studier av tidigare projektspråk, samt referenser till Rubys dokumentation (Britt, 2013).

<3 är ett imperativt programmeringsspråk som har i syntaxen har släktskap med språk såsom Java, Python och XML/HTML. Program som skrivs i <3 läses in från källkodsfiler med ändelsen .three, körs uppifrån och ner och styrs med hjälp av villkor, upprepning och sekvens. <3 är ett interpreterat språk, alltså inte kompilerat. All evaluering sker under körning.

<3 utgörs av en lexer/parser med metoder för att hantera tokens och grammatikregler (*3_parser.rb)*, en fil som beskriver vilka regler och tokens som parsern ska hantera och vad som ska hända när reglerna parsas (*3.rb*), samt en fil som fungerar som en interpretator (*3_tree.rb)*; den består av klassdefinitioner med evalueringsmetoder som används för att bygga upp ett syntaxträd som kan evalueras under körning. Utvecklingen av <3 har skett i programmeringsspråket Ruby 1.9.3 (Ruby-Lang.org, 2013) och språket bör därför också köras i Ruby 1.9.3 för bästa resultat.

Alla konstruktioner som vi avsåg att implementera har implementerats – och fler därtill, men alla konstruktioner fungerar inte som vi ursprungligen avsåg utan modifieringar har skett under utvecklingens gång.

Innehållsförteckning

1 Inledning

1.1 Inledning

Kursen ”TDP019 Projekt: datorspråk” består av ett individuellt projekt på utbildningen Innovativ Programmering på Linköpings universitet som utförs i gemensamt community. Kursen löper parallellt med kursen ”TDP007: Konstruktion av datorspråk”, där teori varvas med praktisk programmering. Projektet pågår under en hel termin (Studiehandboken, 2013).

Den här rapporten beskriver konstruerandet av programmeringsspråket <3 (som matematiskt uttalas "mindre än tre", men vi kort och gott kallar för ”tre”) som har utvecklats av författarna, som ett projekt i kursen TDP019.

1.2 Bakgrund och syfte

Kursens kriterier säger att ett mindre datorspråk ska konstrueras, samt att studenten ska:

- diskutera och motivera designval i det egna datorspråket med utgångspunkt i teori och egna erfarenheter
- implementera verktyg (interpretator, kompilator, etc) för det egna datorspråk [sic]
- formulera teknisk dokumentation av det egna datorspråket

(Studiehandboken, TDP019 Projekt: Datorspråk, 2013)

I ett flertal språk som vi har stött på har vi vid första intrycket förvirrats av hur många olika varianter det finns på att utföra en och samma uppgift i det aktuella språket. Dessa språk har inte känns optimala för oss från det perspektivet att vi varit nybörjare då vi lärt oss dessa.

Vårt syfte utifrån dessa kriterier är att skapa ett datorspråk för nybörjare som vill lära sig programmering. Målet med <3 är därför att begränsa antalet språkliga konstruktioner i så stor grad som möjligt, samt att de bara kan tolkas eller användas på ett eller ett fåtal sätt, men att de tillsammans ändå kan klara de mest grundläggande typer av problemen som man vill lösa med hjälp av ett programmeringsspråk. Sekvens, val, villkor och upprepning implementeras därför.

Vi designar för ändamålet en X/HTML-liknande notation eftersom det tillåter en ingång för de som har varit i kontakt med datorspråk innan, men inte skrivit program. Språket ska vara lätt att lära sig och lätt att skapa program i. Det ska vara entydigt, simpelt och överskådligt i omfång. Vi väljer att skapa ett imperativt språk utan objektorientering, vilket innebär låg svårighetsgrad men hög igenkänningsgrad. Vi väljer att göra språket interpreterat eftersom det tillåter en snabbare och mer dynamisk arbetsprocess. Applikationerna kan tänkas vara script-liknande och exempelvis bäddas in på webben.

Tänkbara ledord för <3 skulle kunna vara: ”En typ av varje typ” och/eller ”En sak – en uppgift”. På så vis blir <3 det hjärtliga språk som titeln på språket (den bildliga eller urbana betydelsen av symbolen <3) antyder.

Man kan tänka sig <3 i bilder av "gränslandet mellan Java, Python och XML".

1.3 Metod och källor

Vi har utgått ifrån föreläsningsanteckningar och lektioner i kurserna TDP007 och TDP019 som teoretisk och praktisk grund i programspråkskonstruktion. För att bygga grammatiken har vi också studerat det imperativa programmeringsspråket Pythons grammatik (Python Software Foundation, 2013), samt tidigare kursprojekt i TDP019 (Linköpings universitet, 2013). Detta har kombinerats med praktisk erfarenhet, tillsammans utövandet av så kallad *trial-and-error* under programmeringsfasen, med mycket refererande till Rubys dokumentation (Britt, 2013). Vi har erhållit en kombinerad parser och lexer från kursen TDP007 och utifrån den kunnat skapa grammatikregler och tokens, och en interpretator. Utvecklingen har skett i programmeringsspråket Ruby, version 1.9.3 (Ruby-Lang, 2013).

2 Systemdokumentation

2.1 Målgrupp

Ett nybörjarspråk alternativt ett övergångsspråk för XML/HTML-programmerare som vill prova på ett imperativt programmeringsspråk men samtidigt känna igen sig i notationen, och till viss del även syntaxen.

2.2 Paradigm och implementering

<3 är ett imperativt programmeringsspråk, som styrs av sekvens, val och upprepning. Programmen i <3 körs uppifrån och ner.

<3 innehåller ej inslag av objektorientering, vilket är någonting som man kan återfinna i andra imperativa språk. Vi har valt att avgränsa dimensionen på språket till enbart den imperativa grunden med motiveringen att språket blir mer överskådligt och endimensionellt för den tänkta målgruppen.

<3 är ett interpreterat språk. Släktskapet med interpreterade språk har påverkat vissa språkliga designval vi har gjort nedan, såsom typing och evaluering.

2.3 Notation och syntax

Block och nyckelord ramas in med taggar, som i ett XML-dokument. <3 är dock ej ett renodlat märkspråk som XML är, <3 är ett programmeringsspråk (jmfr programspråk).

Block kan nästlas, precis som block i t.ex. Python, C++, m.fl.

Inuti blocken kan satser/instruktioner förekomma, som avslutas med semikolon. Taggar kan innehålla attribut, likt XML, för att påverka t.ex. loopars beteende.

Läs mer i bilaga #1: **GRAMMATIK**.

2.4 Program, struktur och sekvens

Program i <3 körs uppifrån och ner (på grund av så kallad *top down parsing*). Måste innehålla (endast) ett programblock som måste inledas med startsymbol (<3>) och avslutas med slutsymbol(</3>), och dessutom innehålla minst en sats.

Exempel 1. Ett program i <3 med en tom sats.

```
<3>
      NULL;
</3>
```

2.5 Konstruktioner i <3

2.5.1 Logik och aritmetik

Vi har valt att implementera fyra räknesätt i grammatiken. Prioritering följer "gängse normer" för aritmetik. Logiska operatorer tillsammans med jämförelseoperatorer kan användas för jämförelser mellan heltal och flyttal (både som literaler och variabler), men inte andra datatyper, av tydlighetsskäl.

Aritmetiska operatorer

Addition	+
Subtraktion	-
Multiplikation	*
Division	/

Dessa operatorer fungerar bara på typerna heltal och flyttal, för avgränsningens skull. På så vis skiljer sig <3 från en del interpreterade språk.

Logiska operatorer och jämförelseoperatorer

Mindre än	<
Mindre än eller lika med	<=
Större än	>
Större än eller lika med	>=
Exakt likhet	==
Icke likhet	!=
Logisk och	&&
Logisk eller	||
Inte	!

2.5.2 Tilldelning

Tilldelnigsperator	=

Vänstersidan tilldelas uttrycket på högersidan. Tilldelning kan ej kedjas, endast en variabel kan tilldelas åt gången.

2.5.3 Räckvidd, omgivning och scope

En variabel har endast den räckvidd som blocket begränsar den till. Vid skymning letas efter närmaste 'x'; interpretatorn letar efter värdet på variabler i närmaste omgivningen först, därefter i den omgivande omgivningen, etc.

Vi har implementerat statisk bindning, som är det vanligast förekommande i imperativa språk.

2.5.4 Variabler

Deklareras och binds i samband med tilldelning. Under körning tilldelas den datatyp som kontexten anger (dynamisk typing), d.v.s `three = 3` tilldelar heltalet 3 till `three` vilket gör det till en heltalstyp, medan tilldelningen `three = "3"`, skapar en `String`. Typkontroll görs under parsning, strängar och heltal är exempelvis inte kompatibla; vi markerar en tydlig avgränsning på det sättet. Däremot ska ett flyttal naturligt kunna konverteras till ett heltal beroende på kontext, och vice versa. <3 är alltså inte helt strikt, utan tillåter viss flexibilitet i typhanteringen.

Överlagring av nyckelord / reserverade ord är ej tillåtet (parsern sätter stopp för det).

Konstanter (kontra variabler) existerar inte i <3.

2.5.5 Datatyper och strukturer, tokens

Nyckelord / reserverade ord: tokens

Identifieras med versaler [A-Z]+. Undantag: start- och slutsymbolerna <3> och </3>. Variabler kan endast deklareras med gemener, så det finns ingen risk att man skriver över något.

Variabler och funktioner: tokens

Gemener och underscore. Endast [a-z_]+, siffror ej tillåtna.

Datatyper: tokens

Siffror [0-9]+ är reserverade för heltal och flyttal.
Sanningsvärden representeras av /TRUE/ *och* /FALSE/.
Strängar ramas enbart in med citationstecken [”] - inte apostrofer ['].

Typing

Dynamisk typing = Binds under körning. Vanligt för scriptspråk, interpreterade språk. Typer finns under ytan, men deklareras aldrig explicit.

Typer

- Heltal (*Integer)*
  Motsvaras av klassen Integer i Ruby.
- Flyttal (*Float*)
  Motsvaras av klassen Float i Ruby.
  Decimalpunkt måste föregås av minst en siffra.
- Sträng (*String*)
  Motsvaras av klassen String i Ruby
  Ramas in med två citationstecken, ett i början, ett i slutet ["] som literaler.
  Escape-sekvens med \ -tecknet gör att man skriva citationstecken inuti en sträng.
  Ett tecken (jmfr *Char* i C++) motsvaras av en sträng med längden 1.

- Sanningsvärde (*Boolean*)
  Kan anta något av värdena TRUE eller FALSE. Motsvaras inte av heltalsvärden, sanningsvärden är enbart av typen *Boolean*. Aritmetiska operationer med sanningsvärden är därför uteslutet.
- Lista (*List*)
  Varje post kan motsvaras av vilken datatyp som helst, även en lista (nästling tillåten). Indexering kan göras på så sätt att man anger med en variabel, literal eller aritmetiskt uttryck som med ett heltal visar vilken post i listan man vill åt.

Typkonvertering
Vi har beslutat att eftersom <3, som riktar sig mot en nybörjarmålgrupp, inte ska tillåta explicit typkonvertering,samtidigt som typer bestäms dynamiskt under körning, det anser vi vara ett designval som kan bidra till förvirring.

Block
Ramas in med `<NYCKELORD>`, till exempel `<IF>` och `</IF>`. Ett block kan innehålla flera block (nästling).

Attribut
Precis som i XML kan man skicka med attribut till vissa typer av konstruktioner (såsom funktioner, loopar och villkorssatser), till exempel kan man säga att en loop ska ha attributet `"WHILE"`.

Separerare
Blanksteg, nyrad och tabulatorsteg tolkas på samma sätt, till skillnad från t.ex. Python. Separerare måste finnas mellan två tokens. Indentering eller tabulatorsteg har ingen syntaktiskt betydelse; satser avslutas med semikolon som kan skrivas flera på rad.

Kommentarer
En typ av kommentar förekommer, liknande den som används i HTML, som kan användas för att kommentera bort en bit av kod, max en rad i taget. Koden innanför kommentarsmarkörerna i exemplet nedan kommer inte att evalueras.

Exempel 2. Kommentar. '<!--' markerar början och '-->' markerar slut på kommentaren.
```
<!-- <3> three = 3; </3> -->
```

Satser
Måste avslutas med semikolon (;). NULL markerar tom sats.
Exempel:
```
      NULL;
      is_a_magic_number = 3;
      three = is_a_magic_number + 0;
```

Villkor
En typ av villkorsats finns: if-satsen. Den tar ett villkor, typ: `<IF magic_number == 3 >` som attribut. Kan nästlas. Tre typer av grenar kan skapas, med nyckelorden: `<IF>, <ELSEIF>, <ELSE>`

Upprepning
Det finns en loop-typ implementerad. Den anpassas med hjälp av attribut (typ av loop), och ett villkor.

Exempel 3. En WHILE-loop i språket <3, följt av PRINT-satser.
```
alla_goda_ting = 0;

<LOOP = "WHILE" alla_goda_ting<3 >
      alla_goda_ting = alla_goda_ting + 1;
</LOOP>

PRINT("Alla goda ting är:");
PRINT(alla_goda_ting);
```

Nyckelord: `<LOOP>, WHILE`

2.5.6 Input/Output (I/O)

I nuvarande version av <3 (1.0) har vi bestämt oss för att implementera en funktion vardera, för I/O.

PRINT
Detta var ursprungligen tänkt som en funktion men är numer snarast en sats som en del av grammatikreglerna för sammansatta satser. Men i källkoden kommer det se ut som ett funktionsanrop där man skickar med vad som ska skrivas innanför parenteserna.

READ
För att använda denna funktion måste man utgå ifrån en tilldelningssats, så att inmatningen sparas undan, sedan måste man komma ihåg att skicka man med en sträng som argument till funktionen, som används uppmaning på skärmen till användaren, innan tangentbordsinmatning.

2.5.7 Funktioner/underprogram

I <3 finns endast en typ av underprogram: funktionen. Den kan antingen returnerar ett värde (”värdet” även kan vara NULL), eller inte returnerar någonting med åstadkommer en "sidoeffekt". Oavsett om man väljer att returnera ett värde eller ej med hjälp av retursats, så kommer funktionen, likt i Ruby, att returnera ett värde, det sista värdet som anges innan funktionens sluttag. Men med en retursats kan man välja att avbryta funktionens körning tidigare.

Funktioner måste vara deklarerade/definierade innan de kan anropas och användas. Attributet `NAME` måste anges, parameterlista (attributet `VAR`) är valfri. Parametrar kan ha standardvärden, alla måste dock adresseras vid anrop.

Parameteröverföringsmodellen är *pass-by-value*, d.v.s. en kopiering av argumentens värden sker inuti i funktionen. Rekursion stöds inte. Överlagring av funktioner stöds ej.

Exempel 4. En funktion med en parameter och en retursats.

```
<FUNCTION NAME= "number_picker" VAR string>
  number = READ(string);
  RETURN number;
</FUNCTION>

magic_number = number_picker("Välj ett nummer");

<IF magic_number == 3 >
  PRINT("is a magic number");
<ELSE>
  PRINT("is not a magic number");
</IF>
```

2.6 Teknisk implementation

<3 är ett interpreterat språk. Programkoden som ska tolkas ska sparas i en fil med ändelsen *.three* i förslagsvis någon standardkodning typ UTF-8. Lexern analyserar sedan teckenströmmen från källkoden, generar tokens som parsern gör en syntaktisk analys av, och ett syntaxträd byggs. Den semantiska analysen, evalueringen av syntaxträdet görs av interpretatorn då programmet körs, antingen går det bra om man har skrivit korrekt kod, eller så avbryts körningen och ett felmeddelande genereras till användaren. Felmeddelanden kan vara av lexikalisk, syntaktisk eller semantisk karaktär; eftersom ingen förkompilering sker, görs all analys under körning.

Miljön som <3 har utvecklats i och följaktligen ska köras i, är programmeringsspråket Ruby, version 1.9.3 (Britt, 2013), för optimalt resultat. Se bilaga #2: **ANVÄNDARHANDLEDNING**, för ytterligare detaljer om hur man skriver och kör program i <3.

2.6.1 Kodstandard

Koden är utvecklad för Ruby 1.9.3-standarden (Britt, 2013). Skulle man välja att exekvera koden i en annan Ruby-miljö kan vi inte garantera full funktionalitet.

Letar man efter en officiell konvention för Rubykodning gör man det förgäves; det finns ingen sådan. Dock skrivs programmeringsspråket Python på ett snarlikt sätt, så vi har använt deras standard – den så kallade “PEP 8 – Style Guide for Python Code” (Python Software Foundation, 2013). PEP är en förkortning för Python Enhancement Proposals. Den förespråkar bland annat konsekvent indententering, `UpperCamelCase` för att namnge klasser, konstanter skrivs med `VERSALER_MED_UNDERSCORE`, andra identifierare med `gemener_separerat_av_underscores`, med mera. Identifierarna har engelska namn som är relevanta för deras syfte. Vi har därtill sett till att kommentera koden där det känns rimligt, men ändå försökt vara sparsamma för att bibehålla läsbarheten (inte ha fler rader kommentarer än kod till exempel). Kommentarerna har dock skrivits på svenska, då målgruppen främst är vår egna IP1-klass, samt examinator.

2.7 Lexikalisk, syntaktisk och semantisk analys

Det som behövs för att kunna konstruera ett datorspråk är dels grammatiska regler, dels en uppsättning med tokens som ska utgöra syntaxen, och dels konstruktioner för analys av rättstavning (lexikalisk analys), analys av grammatiken (syntaktisk analys), bygge av syntaxträd med evalueringsmetoder för att programmet både ska kunna tolkas och exekveras.

Se kod-bilagorna för mer insikt i funktionaliteten.

2.7.1 Lexer och parser

Vi har inte skrivet någon egen parser utan har från kursen TDP007 som laborationsmaterial erhållit filen *rdparse.rb* till hjälp för detta ändamål, som arbetar enligt principen *top down*, *recursive descent*, den jobbar alltså uppifrån och ner med källkoden. *rdparse.rb* har modifierats minimalt, och heter i vårt kodpaket *3_parser.rb*. Den har en lexer-del som matchar och skapar tokens utifrån reguljära uttryck som har definierats i filen *3.rb*, dessa skickas vidare inom *3_parser.rb* till en parser-del för grammatiska regler som definierats i *3.rb*. När parsern matchar reglerna skapas motsvarande objekt, eller noder, för varje konstruktion i språket och på så vis kan ett syntaxträd genereras. När den sista delen i programmet, slutsymbolen, har lästs in, och allt är korrekt skrivet, anropas evalueringsfunktionen i Program-noden, som är skriven i filen *3_tree.rb*, se nästa avsnitt.

2.7.2 Interpretator: syntaxträd med evalueringsmetoder

Syntaxträdet fungerar så att vi har – i princip – en klass för varje konstruktion inuti <3, det vill säga: för varje loop finns ett objekt av klassen `Loop`, för varje heltal finns ett objekt av klassen `Integer`, etcetera. De flesta har också en evalueringsmetod som anropas när hela programmet är parsat, noderna har initierats och eval-funktionen som ligger i Program-objektet körs. Den utlöser en slags kedjereaktion som gör att programmet körs uppifrån och ner precis i samma sekvens som man skrivit det i koden. <3 är ett interpreterat språk, så till skillnad från kompilerade språk kommer vissa fel inte upptäckas förrän under körning, vilket gör att man i eval-funktionerna får lägga in felhantering för sådana situationer. Konstruktionerna för syntaxträdet har implementerats i filen *3_tree.rb*.

Vissa klasser utökar/ärver av redan existerande klasser i Ruby, bland andra String, Integer, och Array. Detta har varit mycket smidigt då man inte behöver göra hela jobbet själv, utan bara det allra nödvändigaste – såsom att lägga till en evalueringsmetod.

2.7.4 Algoritmer

Det enda som krävde algoritmer med någorlunda omfattning var hanteringen av omgivningen; variabeltabellen i vårt språk. Vi hade hela tiden avsikten att implementera ett statiskt scope, men ingen bra lösning på detta själva och studerade därför exempelprojekt från tidigare årgångar av kursen TDP019. Dessa räknas som allmäna resurser, vi har dock valt att markera med kommentarer i koden vad som är lånat från dessa. Den lösning som vi ansåg passade oss bäst återfanns i ett projekt som heter Nibla (Ekberg, 2012), där omgivningen bestod av en klass Scope, som lagrar variablerna i en hashtabell

i flera lager som representerar omgivningar, med en åtkomstmetod med en någorlunda invecklad algoritm som itererar igenom och återskapar de lager som motsvarar rätt omgivning där man vill hämta värdet på den eftersökta variabeln.

Vi försökte modifiera den och anpassa den efter våra behov, men originalkoden fungerade så pass bra att det var ett onödigt bestyr. Däremot kunde vi stryka många rader kod utan att det påverkade funktionaliteten. Originalet fungerar som ett dynamiskt scope, men vår version har statisk kvalitet, tack vara dessa strykningar.

I övrigt finns det inga unika algoritmer i vår kod som kommer att slå världen med häpnad (i alla fall inte som vi uppfattar det).

3 Erfarenheter och reflektion

3.1 Erfarenheter

Vi gick in i projektet ganska nollställda, utan särskilt tydliga riktlinjer eller pekpinnar. Vi hade förkunskaper i Ruby-programmering från TDP007 där vi fick testa att använda en färdigskriven parser. Den visste vi dock inte att vi skulle utgå ifrån när vi började planera projektet i januari, eftersom kurserna löpte parallellt med varandra. Det blev därför rätt mycket gissningsarbete alltmedan kurserna fortlöpte.

Vi trodde t.ex. att vi skulle få hårdkoda mer av parsern själva, när det i själva verket ”bara” krävdes att vi fyllde i rätt tokens i rätt ordning och sedan fyllde i grammatikreglerna och vad de skulle generera för objekt. Vi fick en del övning under Ruby-kursens gång på att göra det, men det visste vi som sagt ingenting om vid språkkonstruktionskursens start när vi skulle planera.

På grund av denna ”nollställdhet” fick vi helt enkelt pröva oss fram, kolla på exempelprojekt och skriva egna minaityrvarianter av språk innan vi fick en känsla av hur vi kunde ta det vidare (*learning by doing*). Till följd av detta bestod första fasen av projektet därför inte alls av kodning, utan att skriva grammatik, vilket i och för sig tog en del tid i anspråk innan den var färdig. Eftersom parsern redan var färdig behövde vi bara fylla i grammatikreglerna och våra tokens däri – det momentet trodde vi skulle vara svårare.

Att hantera och representera omgivningen var oväntat knepigt och en källa till mycken fundering. Vi hade till en början en enkel global hashtabell som representation, men insåg att det inte skulle räcka hela vägen. Vi blev då inspirerade av ett tidigare projekt, som vi nämnt, och skapade klassen Scope, som bygger funktionalitet mot en hashtabell. Vi gjorde lite små ändringar i den som vi trodde var smarta, men som vi fick ändra tillbaka. Likaså utnyttjade vi en lånad algoritm för att söka genom den, den visade sig dock vara överflödig på ett ställe, där den bara ställde till med bekymmer.

Grammatiken var knepig att skriva, vi fick göra om den ett flertal gånger innan vi började skriva koden och även revidera den under implementeringsfasen när vi satte grammatiken på prov. Men nu när vi har jobbat med den så mycket är det lättare att förstå hur formell grammatik fungerar – det gällde 'bara' att komma in i tankesättet – men det tog mycket tid i anspråk. Innan den förståelsen satte sig var vi väldigt beroende av att snegla på tidigare projekt, men med tiden blev vi friare i vårt uttryckssätt.

Att lista ut hur man skulle bygga upp syntaxträdet och evaluera det på rätt sätt och i rätt ordning var också bekymrande till en början, vi fick även där lista ut det mycket på egen hand, men till sist kände vi oss väldigt kreativa i det arbetet, även om det i den första fasen kändes mest som att vi fick chansa oss fram för att någonting överhuvudtaget skulle fungera.

Listor visade sig vara lite jobbigt då vi ett tag trodde att vi hade löst implementationen av dem, men det var då bara parsningen som var implementerad vilket orsakade lite ”panik”, men lyckligtvis var vi inne på rätt spår och det var egentligen bara evalueringen som var problemet; det räckte med att åtgärda det.

Flerradskommentarer däremot, är i skrivande stund ett problem som inte är löst, men enradskommentarer fungerar.

3.2 Reflektion

Att få möjligheten att skapa ett eget datorspråk har lett till många tankar kring hur ett språk är uppbyggt och hur vi själva lär oss nya språk. Även om man intuitivt kan komma långt i att lära sig ett nytt språk, underlättar det alltid att kunna grammatiken som bygger upp språket. Att på detta sätt arbeta med grammatik till ett språk tydliggör varför ett datorspråk fungerar som det gör (och inte fungerar). Ta till exempel arbetet med omgivningen. Här tittade vi igenom andra språk och hur de hanterar detta. Sedan satte vi den informationen i relation till vårt eget språk och vad vi bestämt oss för att implementera. Så här i efterhand hade det varit bra att ha tittat över de språk vi arbetat med under tidigare kurser och utifrån det bildat oss en uppfattning om hur vårt språk skulle vara uppbyggt, innan vi skrev implementationsplanen.

När det gäller vårt språks utseende och hur man skriver för att skapa funktioner, loopar med mera, har vi inte riktigt lyckats skapa en intuitiv känsla att skriva på som vi tänkte från början, och som ett nybörjarspråk bör vara. Dels berodde det på att vi trodde att det skulle bli alltför svårt att skapa ett mer intuitivt språk och vi trodde att det skulle bli tydligare att använda sig av en HTML-liknande notation. Nu känns det i vissa avseenden något rörigt. Om vi fick göra om allt från början igen hade vi nog tänkt lite annorlunda och litat mer på vår egna förmåga, å andra sidan kanske vi inte hade hunnit bli klara i tid om vi hade funderat allt för länge.

När det gäller implementationen så har vi lyckats väldigt bra. Vi har implementerat allt vi har sagt att vi ska implementera och faktiskt mer därtill. Vi har helt enkelt fått lägga till och komplettera grammatiken allteftersom vi har sett att det finns brister och luckor. Vi hade ingen regel för att skapa listor, tilldela listor, och indexera listor. Vi fick grena ut funktionsanrop till att dels vara egen sats och dels kunna stå som en del av en sats. När det gäller I/O-funktionalitet så hade vi inte tänkt att bygga in det i grammatiken men det visade sig vara mycket enklare än vad vi trodde från början. Under arbetets gång har vi möblerat om mycket i de grundläggande reglerna för ”sats-partiklar” såsom PRIMARY (som fick en kusin MATH_PRIMARY), och LITERAL som fick justeras och utökas med BOOL_LITERAL t.ex.

Kommentarfunktionen var tänkt att fungera både för flerrads- och enradskommentarer men fungerar i nuläget enbart för enradsvarianten. Det tror vi beror på att parsern inte läser flera rader i taget.

I mån av tid hade vi avsikten att implementera fler räknesätt, fler datatyper, fler sätt att hantera datatyper (operationer, funktioner, m.m.). Man skulle också kunna tänka sig ut att bygga in större funktionalitet i PRINT-funktionen. Men ”mån av tid” har inte infunnit sig. Ryktet har gått att det finns projekt från tidigare årskurser som har implementerats över en helg. Det har vi svårt att tro på. Vår sammanfattning är att projektet har varit en stor utmaning!

Referenser

Britt, James (2013). *Files, Classes and Methods in Ruby 1.9.3*. <http://ruby-doc.org/core-1.9.3/> . Hämtat under april-maj 2013.

Ruby-Lang.org (2013). *Ruby Programming Language*. <http://www.ruby-lang.org/en/>. Hämtat under april-maj 2013.

Linköpings tekniska högskola (2013). *Studiehandbok@lith*. <http://kdb-5.liu.se/liu/lith/studiehandboken/svkursplan.lasso&k_budget_year=2013&k_kurskod=TDP019> Hämtat 9/5 2013.

Python Software Foundation (2013). *Python Language Reference*. <http://docs.python.org/2/reference/index.html> Hämtat april-maj 2013.

Wallgren, Jonas (2013). Linköpings universitet. *TDP019 Projekt: Datorspråk.* <http://www.ida.liu.se/~TDP019/> . Hämtat under april 2013.

Python Software Foundation (2013). *PEP 8 -- Style Guide for Python Code*. <http://www.python.org/dev/peps/pep-0008/>. Hämtad 8/5 2013.

Ekberg, Albin (2012). Nibla – namnet är omvänt men inte koden. <http://www.ida.liu.se/~TDP019/projekt-2012/NIbla-Dokumentation.pdf> . Hämtad under april 2013.

Bilaga 1. Grammatik för språket <3, version 1.0

Vi använder BNF-notation för att beskriva grammatiken i <3. I övrigt kan förklarande text förekomma under rubrikerna.

Företräde

Dessa prioriteras högst respektive lägst.

```
Subskription
Funktionsanrop
Villkorssatser
Logiska operatorer
Addition & subtraktion
Multiplikation & division
Unära operatorer
```

Program

Startsymbolen är PROGRAM. Ett program måste utgöras av ett <3>-block, som måste innehålla minst en sats och sedan avslutas med </3>. Utförligare specifikation återfinns under respektive rubriker.

```
PROGRAM   ::=  <3> STATEMENTS </3>
```

Aritmetik

Prioritering följer konventionella normer för aritmetiska uttryck.
Multiplikationsoperatorn skapar en produkt av sina argument, som måste vara heltal eller flyttal. Heltal- eller flyttalsdivision bestäms under körning och avgörs av kontexten.
Plus- och minusoperatorerna genererar summor respektive skillnader av sina argument, och fungerar bara på flyttal- och heltalstyperna.
Den unära minusoperatorn generar en negerad högersida. Unära plusoperatorn låter argumentet vara oförändrat.

```
MATH_EXPR       ::=  MULT_EXPR
                |  MATH_EXPR '+' MULT_EXPR
                |  MATH_EXPR '-' MULT_EXPR
MULT_EXPR       ::=  UNARY_EXPR
                |  MULT_EXPR '*' UNARY_EXPR
                |  MULT_EXPR '/' UNARY _EXPR
UNARY_EXPR      ::=  MATH_PRIMARY
                |  '-' MATH_PRIMARY
                |  '+' MATH_PRIMARY
                |  '(' MATH_EXPR ')'
```

Uttryck
Endast gemener och underscore används för identifierare, ej siffor eller andra tecken.
Literaler kan representera heltal, flyttal, sanningsvärden, och strängar. En heltalsliteral kan tolkas som en flyttalsliteral om den står i samma kontext som ett flyttal (addition t.ex.)

Primaryn utgör den/de mest grundläggande beståndsdelarna i språket (icke-satser), atomen är den mest grundläggande delen i ett uttryck; identifierare eller literal.

Uttryck evalueras från vänster till höger, högersidan beräknas dock före vänstersidan.

```
MATH_PRIMARY   ::=  SUBSCRIPTION
                    | FUNC_CALL
                    | IDENTIFIER
                    | NUM_LIT
                    | BOOL_LIT

PRIMARY        ::=  MATH_EXPR
                    | SUBSCRIPTION
                    | FUNC_CALL
                    | ATOM

FUNC_CALL      ::=  IDENTIFIER '()' ';'
                    | IDENTIFIER '(' ARGUMENT ')' ';'
ARGUMENT       ::=  PRIMARY ',' ARGUMENT
                    | PRIMARY

ATOM                ::=  LITERAL
                    | IDENTIFIER
```

Enkla satser och sammansatta satser
I ordning: De mest rudimentära satserna, följt av sammansatta, såsom loopar och villkor. Avslutas med semikolon.
Observera skillnaden mellan FUNC_CALL_STMT (egen sats) och FUNC_CALL (del av uttryck).

```
STATEMENT      ::=  COMPOUND_STMT
                    | ASSIGNMENT_STMT
                    | FUNC_CALL_STMT
                    | BREAK_STMT
                    | RETURN_STMT
                    | NULL_STMT
```

```
COMPOUND_STMT  ::=  IF_STMT
                   | LOOP_STMT
                   | FUNC_DEF
                   | PRINT_STMT
```

Statement list

```
STATEMENTS          ::=  STATEMENT STATEMENTS
                   | STATEMENT
```

Uttryckssatser

Evaluerar logiska uttryck och villkorsuttryck.

```
EXPRESSION          ::=  EXPRESSION '||' AND_TEST
                        AND_TEST

AND_TEST       ::=  AND_TEST '&&' NOT_TEST
                        NOT_TEST

NOT_TEST       ::=  | 'NOT' COMPARISON
                        COMPARISON

COMPARISON      ::=    MATH_PRIMARY '<'  MATH_PRIMARY
                   | MATH_PRIMARY '>'  MATH_PRIMARY
                   | MATH_PRIMARY '==' MATH_PRIMARY
                   | MATH_PRIMARY '>=' MATH_PRIMARY
                   | MATH_PRIMARY '<=' MATH_PRIMARY
                   | MATH_PRIMARY '!=' MATH_PRIMARY
                   | BOOL_LIT
                   | '(' EXPRESSION ')'
```

Tilldelningsatser

Ändrar attribut , binder värden till variabler.

```
ASSIGNMENT_STMT     ::=  TARGET '=' LIST_ASSIGN ';'
                   | TARGET '=' PRIMARY ';'
                   | TARGET '=' EXPRESSION ';'
                   | TARGET '=' READ_FUNC ';'

TARGET         ::=  SUBSCRIPTION
                   | IDENTIFIER
```

```
LIST_ASSIGN    ::=  '{' LIST_VALUES '}'
                  | '{' '}'

LIST_VALUES    ::=  LIST_VALUE ',' LIST_VALUES
                  | LIST_VALUE

LIST_VALUE          ::=  LIST_ASSIGN
                  | EXPRESSION
                  | PRIMARY
```

Indexering

```
SUBSCRIPTION   ::=  STRING_LIT '[' SUB ']'
                  | IDENTIFIER '[' SUB ']'
                  | FUNC_CALL '[' SUB ']'

SUB            ::=  MATH_EXPR ']' '[' SUB
                  | MATH_EXPR
```

Retursatsen

Returnerar ett värde eller null från en funktion.

```
RETURN_STMT ::= 'RETURN' ( NULL_STMT | PRIMARY | EXPRESSION |
LIST_ASSIGN) ';'
```

Breaksatsen

Avbryter en loop.

```
BREAK_STMT           ::= 'BREAK' ';'
```

Nullsatsen

Gör ingenting. Används där en sådan sats krävs av syntaxen.

```
NULL_STMT      ::= 'NULL' ';'
```

Villkorssats

```
IF_STMT         ::=  IF_PART ELSEIF_PART ELSE_PART '</IF>'
                     | IF_PART ELSE_PART '</IF>'
                     | IF_PART ELSEIF_PART '</IF>'
                     | IF_PART '</IF>'
IF_PART         ::=  '<' 'IF' EXPRESSION '>' STATEMENTS
ELSEIF_PART           ::=  ELSEIF_PART, ELSEIF
                     | ELSEIF
ELSEIF          ::=  | '<' 'ELSEIF' EXPRESSION '>' STATEMENTS
ELSE_PART       ::=  '<' 'ELSE' '>'  STATEMENTS
```

Loopsats

```
LOOP_STMT       ::=  '<' 'LOOP' '=' 'WHILE' EXPRESSION '>'
                      STATEMENTS
                     '</LOOP>'
```

Funktioner

```
FUNC_DEF        ::=  '<' 'FUNCTION' 'NAME' '=' '"' IDENTIFIER '"'
                     PARAMETERS '>' STATEMENTS '</FUNCTION>'

                     | '<' 'FUNCTION' 'NAME' '=' '"' IDENTIFIER '"'
'>'
                       STATEMENTS '</FUNCTION>'


PARAMETERS           ::=  PARAMETER "," PARAMETERS
                     | PARAMETER

PARAMETER       ::=  'VAR' IDENTIFIER '=' PRIMARY
                     | 'VAR' IDENTIFIER
```

I/O

Användarstyrd in- och utmatning via tangentbord och skärm.

```
PRINT_STMT           ::=  'PRINT' '(' PRIMARY ')' ';'
READ_FUNC       ::=  'READ' '(' IDENTIFIER ')'
                     | 'READ' '(' STRING_LIT ')'
```

Nyckelord / Reserverade ord
Är egentligen reserverade per automatik i och med att de skrivs versalt.

```
BREAK
ELSE
ELSEIF
IF
FUNCTION
NAME
VAR
LOOP
WHILE
NULL
RETURN
TRUE
FALSE
```

Bilaga 2. Användarhandledning

Fyra steg för att komma igång och skriva ditt första program

1. Ladda ner Ruby version 1.9.3. Anledning till att just den här versionen krävs är att interpretatorn och parsern för språket <3 är skriven i programmeringsspråket Ruby, för just den versionen. Det finns mycket bra information på Rubys officiella hemsida http://www.ruby-lang.org. Om du har Ubuntu kan du via terminalfönstret ladda ner ruby genom att skriva:

```
$ sudo apt-get install ruby1.9.3
```

   Kom ihåg att du måste packa upp alla filer i installationsmappen till samma mapp, de du behöver är: *3.rb*, *3_parser.rb* och *3_tree.rb*.

2. Skapa en fil med filändelsen *three*, till exempel *mittProgram.three*.

3. I filen skriver du följande (glöm inte att spara!). Funktionen `PRINT` kommer du att ha mycket nytta av, när du ska spåra vad som händer i dina program.

```
<3>
   PRINT("Hello world!");
</3>
```

4. Öppna upp ett terminalfönster där du kan köra ditt program genom att skriva:

```
$ ruby 3.rb
```

   <3 frågar efter filen du vill köra:

```
Welcome to <3!
Please specify the source code file to be interpreted (ex: 3.three):
```

   då skriver du i detta fallet `mittProgram.three` och trycker på <retur>.

Nu har du skrivit ditt första program i <3!

Det är viktigt att komma ihåg att grammatiken i <3 kräver att du har minst en sats i programmet innanför start- och slutsymbolerna (`<3` och `</3>`), den enklaste satsen är `NULL;` som är en tom sats. En sats avslutas alltid med semikolon!

Konstruktioner och byggstenar i <3

Ett programmeringsspråk består av konstruktioner och byggstenar med vilkas hjälp du kan lösa problem eller utföra uppgifter som du kanske vill slippa göra själv. Sekvens, selektion och upprepning är typexempel på konstruktioner som löser vilket godtyckligt problem som helst. Utöver det behöver du partiklar såsom variabler och funktioner som du kan använda i dina konstruktioner. När du sedan ska beordra ditt program att använda konstruktionerna på rätt sätt behöver du skriva grammatiskt korrekt syntax, fördjupning inom det området görs lämpligast genom att läsa bilaga #1: **GRAMMATIK**.

Variabler

En variabel kan beskrivas som en behållare av information, ett sätt att identifiera värden och spara dem I minnet. Språket <3 gör ingen skillnad på olika typer av variabler, dvs du behöver inte skriva (deklarera) vilken typ variabeln ska vara, det gör språket åt dig. I språket <3 finns fem olika typer av variabler; heltal, flyttal, strängar, sanningsvärden och specialfallet lista som kan innehålla alla de andra datatyperna inklusive "sig själv". Exempel på ett heltal är `3`, ett flyttal `3.3`, en sträng: `"three"` och ett sanningsvärde: `TRUE`.

Till att börja med väljer du ett variabelnamn, exempelvis `text`. Tilldela därefter variabeln ett värde, med hjälp av en tilldelningssats. I det här fallet värdet `"Hello world!"`. Med hjälp av funktionen `PRINT` kan du skriva ut värdet (i det här fallet en sträng) på skärmen.

```
<3>
  text = "Hello world!";
  PRINT(text);
</3>
```

Skriv, spara och kör programmet (se punkt 4 ovan). Nu kommer det att se ut så här i terminalfönstret:

```
Hello world!
```

Prova att ändra variabelns värde och se vad som händer.

Listor är en lite speciell typ av variabel som gör att du kan spara många värden på ett och samma ställe, i samma behållare. Säg att du vill hålla reda på tre saker som har något gemensamt, då skriver du så här:

```
frukost = {"fil", "treo", "juice"};
```

De olika värdena i en lista kan hämtas ut genom så kallad indexering. Det innebär att den första positionen i listan, i det här fallet `"fil"`, har position `0`. För att exempelvis skriva ut `"treo"` från listan skriver du

```
PRINT(frukost[1]);
```

Funktioner

En funktion är en del av ett datorprogram som kan anropas för att utföra en viss uppgift oberoende av resten av koden. En funktion utformas så att de kan anropas flera gånger från olika ställen i programmet och då skickar man ofta med argument till funktionen, som lägger in värdena på argumenten i dess parameterlista. En funktion kan sedan arbeta med värdena och även returnera ett värde till platsen där den anropades. Låter det krångligt? Vi tar det steg för steg.

Säg att vi vill skapa en funktion som räknar ut summan av två tal och returnerar värdet.
Först ger vi funktionen ett namn, exempelvis “sum” och namnen på två parametrar (`VAR a` och `b`)
Observera att parametrarna inte har något värde än, värdena kommer du att skicka in som argument när du kallar på funktionen (vi kommer dit snart).

```
<3>
      <FUNCTION NAME = "sum" VAR a, VAR b >

      </FUNCTION>
</3>
```

Inuti funktionen ska du nu räkna ut summan av parameter a och parameter b, samt returnera summan.

```
<3>
      <FUNCTION NAME = "sum" VAR a, VAR b >
        result = a+b;
        RETURN result;
      </FUNCTION>
</3>
```

Nu återstår det bara att kalla på funktionen. Beroende på vilka värden du skickar in till funktionen, kommer olika värden att returneras. För att returvärdet ska sparas måste du ha en variabel tillgänglig för det, då sätter du helt enkelt funktionsanropet i en tilldelningssats, istället för att låta funktionsanropet stå som egen sats (i det fallet vill du troligtvis bara ha en sidoeffekt, såsom utskrift).

```
<3>
      <FUNCTION NAME = "sum" VAR a, VAR b >
       result = a+b;
       RETURN result;
      </FUNCTION>

      magic_number = sum(2,1);
      PRINT(magic_number);
</3>
```

När du kör det här programmet i terminalfönstret kommer det här att skrivas ut:

```
3
```

Prova att ändra uttrycket i funktionen så att funktionen räknar ut produkten av två tal istället. Prova även att skicka in olika tal.

If-sats / villkorssats

En if-sats är en sats som bara körs om ett villkor är uppfyllt, man skulle kunna beskriva det som att “om det här villkoret är sant så gör det här”. I språket <3 finns följande operatorer för att skapa villkor:

&&	OCH
||	ELLER
NOT	inte

==	lika med
>=	större eller lika med
<=	mindre eller lika med
!=	inte lika med
<	mindre än
>	större än

Exempelvis är uttrycket 3 == 3 sant, medan 3 == 2 är falskt. Däremot är 3 != 2 sant. Uttycket 2 < 3 är sant, medan 2 > 3 är falskt.

I det här meningsfulla programmet vill vi skriva ut en text om 3 är lika med 3 (vilket ju alltid är sant).

```
<3>
    <IF 3 == 3>
        PRINT("3 is a magic number");
    </IF>
</3>
```

Du kan lägga till fler än ett villkor. Säg att du har två stycken variabler med värdena 2 och 3 och vill att programmet skriver ut vilket värde som är störst.

```
<3>
      two = 2;
      three = 3;

      <IF two < three >
            PRINT("2 is <3");
      <ELSEIF NOT two < three >
            PRINT("2 is not <3");
```

Prova att ändra värdena på variablerna. Exempelvis `two = 3;` och `three = 2;`

Loop / upprepning

Loopar används för att åstadkomma en upprepning av en sats. Tänk dig till exempel att du vill skriva ut talet 3 13 gånger, vilket du förmodligen har stor nytta av. Du skulle kunna skriva så här:

```
<3>
      PRINT("3");
      PRINT("3");
```

```
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
        PRINT("3");
    </3>
```

Du kan givetvis skicka talet `3` istället för strängen `"3"` som argument till `PRINT` om du vill.

Men hur blir det om du vill skriva ut 3 333 gånger eller 3333 gånger? Då är det smart att använda sig av en loop. Språket <3 har en typ av loop som heter `"WHILE".` Du behöver ange ett villkor för loopen, i det här fallet `i < 13` vilket betyder att så länge `i` är mindre än 13 så skrivs `3` ut. Det är viktigt att räkna upp variabeln `i` inuti loopen, annars skapas en oändlig loop, eftersom `i` i sådana fall alltid kommer att vara 0 och mindre än 13.

```
    <3>
        i = 0;
        <LOOP = "WHILE" i<13 >
            i = i+1;
            PRINT(3);
        </LOOP>
    </3>
```

Prova att ange villkoret `i < 333` som villkor för loopen istället.

Kommentarer

Det som skrivs mellan kommentartecknen kommer att utgöra ett stycke källkod som inte kommer att tolkas av språket <3. Koden körs alltså inte som vanlig kod utan kommer att hoppas över. Kommentarer är bra då du vill kommentera vad ett kodstycke gör eller när du vill felsöka, då du kan "plocka bort" viss kod från körningen. Kommentarer skrivs så här: `<!-- kommentar här tack -->` Det är till exempel god sed att skriva vem som skapat programmet och vad programmet ska göra.

```
    <3>
        <!-- Författare: Hannah & Per -->
        <!-- Ett litet program som räknar ut produkten av två tal -->
        <!-- PRINT(3 + 3); -->

        PRINT(3 * 3);
    </3>
```

Programmet kommer att skriva ut `9` och ignorera allt som är kommenterat.

I/O

PRINT och READ

In- och utmatning kommer du att ha mycket nytta av i dina program. Hittills har du bara använt funktionen/satsen `PRINT` – som skriver ut värden av variabler och literaler på skärmen – när du velat interagera med användaren. Du kan också läsa indata från tangentbordet; låt säga att du vill skapa ett program som instruerar användaren att skriva in sitt namn och sedan låta programmet skriva ut namnet på skärmen.

```
      <3>
            namn = READ("Skriv ditt namn");
            PRINT("Hej")
            PRINT(namn);
      </3>
```

Du kan skapa ett program som tar in två siffror och skriver ut summan av dessa.

```
      <3>
            a = READ("Skriv in ett tal:");
            b = READ("Skriv in ett tal till:");
            PRINT("Summan av de talen du matade in blir:");
            PRINT(a+b);
      </3>
```

Här tilldelas variablerna `a` och `b` talen som matas in av användaren, och skrivs sedan ut.

Break / avbryt

Det lilla nyckelordet/satsen `BREAK` används när du vill avbryta någon del av körningen, låt säga att du vill avbryta en loop när ett visst villkor är uppfyllt. Nedan ska vi kombinera det tricket med det mesta av det vi lärt oss hittills i ett litet nonsensprogram.

```
<3>

  <FUNCTION NAME = "foo" >
    loop = TRUE;

    <LOOP = "WHILE" loop == TRUE>
      input = READ("Mata in ett tal, inte <3, helst 3!");

      <IF input <3 >
        PRINT("Inte <3, sa jag!");
        BREAK;
```

```
      <ELSEIF input > 3 >
        PRINT("Testa igen!");
      <ELSE>
        PRINT("Bra!");;
        RETURN 3;
      </IF>
    </LOOP>
  </FUNCTION>

  bar = foo();
  PRINT(bar);

</3>
```

Grattis! Du är nu utrustad med de verktyg som krävs för att kunna skriva dina egna 3-vliga program.

Felmeddelanden i <3

Ibland går det inte som man tänkt sig. Här följer några exempel på felmeddelanden som kan dyka upp när du försöker köra din kod, och vad du i sådana fall har gjort för fel.

Fel #1

```
Parse error. Expected: […] found …
```

Varje gång du får detta felmeddelande, kan det bero på en mängd olika orsaker och det finns varierande utseende på detta felmeddalnde beroende I vilket sammanhang felet har uppstått. Du har antingen gjort en felstavning (lexikaliskt fel) eller så har skrivit fel grammatik (syntaxfel). Det kan då röra sig om att du glömt att skriva semikolon efter en sats för att avsluta den (vanligt fel), att du försöker slå ihop variabler av olika datatyper (t.ex: `1 + "3",` vilket inte stöds av grammatiken), att du glömt att avsluta ett block med `</`, att du har glömt åtskilja parametrar I en parameterlista med kommatecken, etcetera, exemplen är många fler än vad som får plats här.

Fel #2

```
<3 Name Error: The variable a does not exist
```

Du har glömt att tilldela variabeln du letar efter, eller så ligger variabeln i en omgivning som inte är tillgänglig.

Fel #3

```
Argument Error: Wrong number of arguments
```

Du skickar in för få eller för många argument till funktionen, kolla parameterlistan till funktionen.

Fel #4

```
<3 Assign Error: Target is not subscriptable
```

Du försöker indexera en variabel som inte är en lista och göra en tilldelning.

```
            <ELSE>
                  PRINT("Your logic is flawed");
            </IF>
      </3>
```

När du kör det här programmet kommer utskriften att bli:

```
2 is <3
```

Prova att ändra värdena på variablerna. Exempelvis `two = 3;` och `three = 2;`

Loop / upprepning

Loopar används för att åstadkomma en upprepning av en sats. Tänk dig till exempel att du vill skriva ut talet 3 13 gånger, vilket du förmodligen har stor nytta av. Du skulle kunna skriva så här:

```
      <3>
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
            PRINT("3");
      </3>
```

Du kan givetvis skicka talet `3` istället för strängen `"3"` som argument till `PRINT` om du vill.

Men hur blir det om du vill skriva ut 3 333 gånger eller 3333 gånger? Då är det smart att använda sig av en loop. Språket <3 har en typ av loop som heter `"WHILE".` Du behöver ange ett villkor för loopen, i det här fallet `i < 13` vilket betyder att så länge `i` är mindre än 13 så skrivs `3` ut. Det är viktigt att räkna upp variabeln `i` inuti loopen, annars skapas en oändlig loop, eftersom `i` i sådana fall alltid kommer att vara 0 och mindre än 13.

```
      <3>
            i = 0;
            <LOOP = "WHILE" i<13 >
                  i = i+1;
```

```
            PRINT(3);
        </LOOP>
    </3>
```

Prova att ange villkoret `i < 333` som villkor för loopen istället.

Kommentarer

Det som skrivs mellan kommentartecknen kommer att utgöra ett stycke källkod som inte kommer att tolkas av språket <3. Koden körs alltså inte som vanlig kod utan kommer att hoppas över. Kommentarer är bra då du vill kommentera vad ett kodstycke gör eller när du vill felsöka, då du kan "plocka bort" viss kod från körningen. Kommentarer skrivs så här: `<!-- kommentar här tack -->` Det är till exempel god sed att skriva vem som skapat programmet och vad programmet ska göra.

```
    <3>
        <!-- Författare: Hannah & Per -->
        <!-- Ett litet program som räknar ut produkten av två tal -->
        <!-- PRINT(3 + 3); -->

        PRINT(3 * 3);
    </3>
```

Programmet kommer att skriva ut `9` och ignorera allt som är kommenterat.

I/O

PRINT och READ

In- och utmatning kommer du att ha mycket nytta av i dina program. Hittills har du bara använt funktionen/satsen `PRINT` – som skriver ut värden av variabler och literaler på skärmen – när du velat interagera med användaren. Du kan också läsa indata från tangentbordet; låt säga att du vill skapa ett program som instruerar användaren att skriva in sitt namn och sedan låta programmet skriva ut namnet på skärmen.

```
    <3>
        namn = READ("Skriv ditt namn");
        PRINT("Hej")
        PRINT(namn);
    </3>
```

Du kan skapa ett program som tar in två siffror och skriver ut summan av dessa.

```
    <3>
        a = READ("Skriv in ett tal:");
        b = READ("Skriv in ett tal till:");
```

```
            PRINT("Summan av de talen du matade in blir:");
            PRINT(a+b);
      </3>
```

Här tilldelas variablerna `a` och `b` talen som matas in av användaren, och skrivs sedan ut.

Break / avbryt

Det lilla nyckelordet/satsen `BREAK` används när du vill avbryta någon del av körningen, låt säga att du vill avbryta en loop när ett visst villkor är uppfyllt. Nedan ska vi kombinera det tricket med det mesta av det vi lärt oss hittills i ett litet nonsensprogram.

```
<3>

  <FUNCTION NAME = "foo" >
    loop = TRUE;

    <LOOP = "WHILE" loop == TRUE>
      input = READ("Mata in ett tal, inte <3, helst 3!");

      <IF input <3 >
        PRINT("Inte <3, sa jag!");
        BREAK;
      <ELSEIF input > 3 >
        PRINT("Testa igen!");
      <ELSE>
        PRINT("Bra!");;
        RETURN 3;
      </IF>
    </LOOP>
  </FUNCTION>

  bar = foo();
  PRINT(bar);

</3>
```

Grattis! Du är nu utrustad med de verktyg som krävs för att kunna skriva dina egna 3-vliga program.

Felmeddelanden i <3

Ibland går det inte som man tänkt sig. Här följer några exempel på felmeddelanden som kan dyka upp när du försöker köra din kod, och vad du i sådana fall har gjort för fel.

Fel #1

```
Parse error. Expected: […] found …
```

Varje gång du får detta felmeddelande, kan det bero på en mängd olika orsaker och det finns varierande utseende på detta felmeddalnde beroende I vilket sammanhang felet har uppstått. Du har antingen gjort en felstavning (lexikaliskt fel) eller så har skrivit fel grammatik (syntaxfel). Det kan då röra sig om att du glömt att skriva semikolon efter en sats för att avsluta den (vanligt fel), att du försöker slå ihop variabler av olika datatyper (t.ex: `1 + “3”,` vilket inte stöds av grammatiken), att du glömt att avsluta ett block med `</`, att du har glömt åtskilja parametrar I en parameterlista med kommatecken, etcetera, exemplen är många fler än vad som får plats här.

Fel #2

```
<3 Name Error: The variable a does not exist
```

Du har glömt att tilldela variabeln du letar efter, eller så ligger variabeln i en omgivning som inte är tillgänglig.

Fel #3

```
Argument Error: Wrong number of arguments
```

Du skickar in för få eller för många argument till funktionen, kolla parameterlistan till funktionen.

Fel #4

```
<3 Assign Error: Target is not subscriptable
```

Du försöker indexera en variabel som inte är en lista och göra en tilldelning.

Bilaga 3. 3.rb

```ruby
#!/usr/bin/env ruby -w
# -*- coding: utf-8 -*-

# Importera lexer, parser, evaluator
require './3_parser'
require './3_tree'

############################## 3.rb ##############################################
#                                                                                #
# <3 by Hannah Börjesson and Per Jonsson 2013                                    #
# @Innovativ Programmering, Linköping university                                 #
#                                                                                #
# A lovable interpreted imperative programming language                          #
#                                                                                #
# This file consists of tokens and grammar rules                                 #
#                                                                                #
# The rest will follow in Swedish                                                #
#                                                                                #
##################################################################################

class LessThanThree
  def initialize
    @threeParser = Parser.new(" <3 ") do

      ##### ###### #####
      ##### TOKENS #####
      ##### ###### #####

      # Namnen på tokens torde förklara deras syften #

      token( /<!-+.*-+>/)#/m)

      token(/^"[^\"]*"/) {|str_lit| str_lit.to_s}

      token(/\s+/) # ignorera blanksteg
      token(/\t+/) # och tabbar
```

```
token(/<3>/) {|start| :BEGIN}
token(/<\/3>/) {|_end_| :END}

token(/NULL/) {|null_stmt| null_stmt}
token(/BREAK/) {|break_stmt| break_stmt}
token(/RETURN/) {|return_stmt| return_stmt}

token(/(\[|\])/) {|subscript| subscript}

token(/<\/IF>/) {|end_if| end_if}
token(/IF/) {|_if_| _if_}
token(/ELSEIF/) {|elseif| elseif}
token(/ELSE/) {|_else_| _else_}

token(/<\/LOOP>/) {|end_loop| end_loop}
token(/LOOP/) {|loop| loop}
token(/WHILE/) {|_while_| _while_}

token(/<\/FUNCTION>/) {|end_func| end_func}
token(/FUNCTION/) {|function| function}
token(/NAME/) {|name| name}
token(/VAR/) {|var| var}

token(/^[a-z_]+/) {|identifier| identifier}
token(/\{/) {|list_left| list_left}
token(/\{/) {|list_right| list_right}

token(/\(\)/) {|func_call| func_call}

token(/PRINT/) {|print| print}
token(/READ/) {|read| read}

token(/TRUE/) {|_true_| :TRUE}
token(/FALSE/) {|_false_| :FALSE}

token(/-?\d+\.\d+/) {|num_lit| num_lit.to_f}
token(/-?\d+/) {|num_lit| num_lit.to_i}

token(/\|\|/) {|_or_| _or_}
token(/&&/) {|_and_| _and_}
```

```
token(/NOT/) {|_not_| _not_ }

token(/==/) {|comp_op| comp_op}
token(/>=/) {|comp_op| comp_op}
token(/<=/) {|comp_op| comp_op}
token(/!=/) {|comp_op| comp_op}
token(/</) {|comp_op| comp_op}
token(/>/) {|comp_op| comp_op}

token(/(\+|-|\*|\/)/) {|math_op| math_op};

token(/=/) {|assign| :assign }

token(/;+/) {|end_stmt| :end_stmt}

token(/./) {|match| match}


##### ############### #####
##### GRAMMATIKREGLER #####
##### ############### #####

# Regler matchas,för varje matchning byggs ett syntaxträd upp
# som evalueras när vi kommit till toppnoden Program

# Full grammatikförteckning med förklaringar finns i projektdokumentationen

start :PROGRAM do
  match(:BEGIN, :STATEMENTS, :END) {|_, statements, _| Program.new(statements).eval}
end

rule :STATEMENTS do
  match(:STATEMENT, :STATEMENTS)  {|stmt, stmts| Statements.new(stmt, stmts)}
  match(:STATEMENT) {|stmt| Statements.new(stmt)}
end

rule :STATEMENT do
  match(:COMPOUND_STMT)
  match(:ASSIGNMENT_STMT)
  match(:FUNC_CALL_STMT)
```

```
  match(:BREAK_STMT)
  match(:RETURN_STMT)
  match(:NULL_STMT)
end

rule :NULL_STMT do
  match('NULL', :end_stmt){Null.new}
end

rule :BREAK_STMT do
  match('BREAK',:end_stmt){Break.new}
end

rule :RETURN_STMT do
   match('RETURN',:NULL_STMT) {|_,null| Return.new(null)}
   match('RETURN',:PRIMARY, :end_stmt) {|_,primary,_| Return.new(primary)}
   match('RETURN',:EXPRESSION, :end_stmt) {|_,expr,_| Return.new(expr)}
   match('RETURN',:LIST_ASSIGN, :end_stmt) {|_,expr,_| Return.new(expr)}
end

rule :ASSIGNMENT_STMT do
  match(:TARGET,:assign,:LIST_ASSIGN, :end_stmt) {|target,_,list,_| Assign.new(target, list)}
  match(:TARGET,:assign,:PRIMARY,:end_stmt) {|target,_,prim,_| Assign.new(target, prim)}
  match(:TARGET,:assign,:EXPRESSION,:end_stmt) {|target,_,expr,_| Assign.new(target, expr)}
  match(:TARGET,:assign,:READ_FUNC,:end_stmt) {|target,_,input,_| Assign.new(target, input)}
end

rule :TARGET do
  match(:SUBSCRIPTION)
  match(:IDENTIFIER)
end

rule :LIST_ASSIGN do
  match('{',:LIST_VALUES,'}') {|_,list_values,_| List.new(list_values)}
  match('{', '}') {|_, _| List.new(nil)}
end

rule :LIST_VALUES do
  match(:LIST_VALUE,',',:LIST_VALUES) {|value,_,values| [values] + [value] }
  match(:LIST_VALUE) {|value| value}
```

```
    end

    rule :LIST_VALUE do
      match(:LIST_ASSIGN)
      match(:EXPRESSION)
      match(:PRIMARY)
    end

    rule :FUNC_CALL_STMT do
      match(:IDENTIFIER,'()',:end_stmt) {|id,_,_| FunctionCall.new(id)}
      match(:IDENTIFIER,'(',:ARGUMENT,')',:end_stmt) {|id,_,args,_,_| FunctionCall.new(id, args)}
    end

    rule :ARGUMENT do
      match(:PRIMARY,',',:ARGUMENT) {|arg,_,args| ArgumentList.new(arg, args)}
      match(:PRIMARY) {|arg| ArgumentList.new(arg)}
    end

    rule :FUNC_CALL do
      match(:IDENTIFIER,'()') {|id,_,_| FunctionCall.new(id)}
      match(:IDENTIFIER,'(',:ARGUMENT,')') {|id,_,args,_,_| FunctionCall.new(id, args)}
    end

    rule :COMPOUND_STMT do
      match(:IF_STMT)
      match(:LOOP_STMT)
      match(:FUNC_DEF)
      match(:PRINT_STMT)
    end

    rule :IF_STMT do
      match(:IF_PART,:ELSEIF_PART,:ELSE_PART,'</IF>') {|if_p,elseif_p,else_p,_| IfStmt.new(if_p,
elseif_p, else_p)}
      match(:IF_PART,:ELSEIF_PART,'</IF>') {|if_p,elseif_p,_| IfStmt.new(if_p, elseif_p) }
      match(:IF_PART,:ELSE_PART,'</IF>') {|if_p,else_p,_| IfStmt.new(if_p, nil, else_p)}
      match(:IF_PART,'</IF>') {|if_p,_| IfStmt.new(IfStmt.new(if_p))}
    end

    rule :IF_PART do
      match('<', 'IF', :EXPRESSION,'>',:STATEMENTS) {|_,_,expr,_,stmts| If.new(stmts, expr)}
    end
```

```
      rule :ELSEIF_PART do
        match(:ELSEIF_PART, :ELSEIF)  {|part, elseif| ElseIfs.new(part, elseif)}
        match(:ELSEIF) {|elseif| ElseIfs.new(elseif)}
      end

      rule :ELSEIF do
        match('<', 'ELSEIF',:EXPRESSION,'>',:STATEMENTS) {|_,_,expr,_,stmts| If.new(stmts, expr)}
      end

      rule :ELSE_PART do
        match('<', 'ELSE', '>',:STATEMENTS) {|_,_,_,stmts| If.new(stmts)}
      end

      rule :LOOP_STMT do
        match('<', 'LOOP',:assign, '"WHILE"',:EXPRESSION,'>',:STATEMENTS,'</LOOP>')
{|_,_,_,_,expr,_,stmts,_| Loop.new(expr, stmts)}
      end

      rule :EXPRESSION do
        match(:EXPRESSION,'||',:AND_TEST) {|expr,op,and_test| Expression.new(expr,"or",and_test)}
        match(:AND_TEST)
      end

      rule :AND_TEST do
        match(:AND_TEST,'&&',:NOT_TEST) {|and_test,_,not_test| Expression.new(and_test,"and",not_test)}
        match(:NOT_TEST)
      end

      rule :NOT_TEST do
        match('NOT',:COMPARISON) {|_,comparison| Expression.new(comparison, "not")}
        match(:COMPARISON)
      end

      rule :COMPARISON do
        match(:MATH_PRIMARY,'==',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, "==", p2)}
        match(:MATH_PRIMARY,'>=',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, ">=", p2)}
        match(:MATH_PRIMARY,'<=',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, "<=", p2)}
        match(:MATH_PRIMARY,'!=',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, "!=", p2)}
        match(:MATH_PRIMARY,'<',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, "<", p2)}
        match(:MATH_PRIMARY,'>',:MATH_PRIMARY) {|p1,_,p2| Comparison.new(p1, ">", p2)}
```

```
    match(:BOOL_LIT) {|b| Comparison.new(b)}
    match('(',:EXPRESSION, ')') {|_,expr,_| expr}
  end

  rule :SUBSCRIPTION do
    match(:STRING_LIT, '[', :SUB, ']') {|str,_,index,_| Subscription.new(str, index)}
    match(:IDENTIFIER, '[', :SUB, ']')  {|id,_,index,_| Subscription.new(id, index)}
    match(:FUNC_CALL, '[', :SUB, ']') {|func,_,index,_| Subscription.new(func, index) }
  end

  rule :SUB do
    match(:MATH_EXPR, ']', '[', :SUB) {|m,_,_,s| SubValues.new(m,s)}
    match(:MATH_EXPR)
  end

  rule :FUNC_DEF do
    match('<', 'FUNCTION', 'NAME',:assign, String, :PARAMETERS,'>', :STATEMENTS,'</FUNCTION>')
{|_,_,_,_,id,pars,_,stmts,_|  Function.new(id[1, id.size-2], stmts, pars)}
    match('<', 'FUNCTION', 'NAME', :assign, String,'>',:STATEMENTS,'</FUNCTION>')
{|_,_,_,_,id,_,stmts,_| Function.new(id[1, id.size-2], stmts) }
  end

  rule :PARAMETERS do
    match(:PARAMETER,',',:PARAMETERS) {|par,_,pars| ParameterList.new(par, pars) }
    match(:PARAMETER)
  end

  rule :PARAMETER do
    match('VAR',:IDENTIFIER,:assign,:PRIMARY) {|_,name,_,value| Parameter.new(name, value)}
    match('VAR',:IDENTIFIER) {|_, name| Parameter.new(name)}
  end

  rule :PRINT_STMT do
    match('PRINT','(', :PRIMARY ,')', :end_stmt) {|_,_,p,_,_| Print.new(p)}
  end

  rule :READ_FUNC do
    match('READ','(', :IDENTIFIER ,')') {|_,_,id,_,_| Read.new(id)}
    match('READ','(', :STRING_LIT ,')') {|_,_,str,_,_| Read.new(str)}
  end
```

```
      rule :PRIMARY do
        match(:MATH_EXPR)
        match(:SUBSCRIPTION)
        match(:FUNC_CALL)
        match(:ATOM)
      end

      rule :ATOM do
        match(:IDENTIFIER)
        match(:LITERAL)
      end

      rule :LITERAL do
        match(:STRING_LIT)
        match(:BOOL_LIT)
        match(:NUM_LIT)
      end

      rule :STRING_LIT do
        match(String)
        end

      rule :NUM_LIT do
        match(Integer) {|integer| NumLit.new(integer) }
        match(Float) {|float| NumLit.new(float) }
      end

      rule :BOOL_LIT do
        match(:TRUE) { BoolLit.new(true) }
        match(:FALSE) { BoolLit.new(false) }
      end

      rule :IDENTIFIER do
        match(/^[a-z_]+/) {|identifier| Variable.new(identifier)}
      end

      rule :MATH_EXPR do
        match(:MATH_EXPR, '+', :MULT_EXPR) {|math_expr,op,mult_expr| Mathreematics.new(math_expr, op,
mult_expr)}
        match(:MATH_EXPR, '-', :MULT_EXPR) {|math_expr,op,mult_expr| Mathreematics.new(math_expr, op,
mult_expr)}
```

```
        match(:MULT_EXPR)
        end

      rule :MULT_EXPR do
        match(:MULT_EXPR,'*', :UNARY_EXPR) {|mult_expr,op,unary_expr| Mathreematics.new(mult_expr, op,
unary_expr)}
        match(:MULT_EXPR,'/', :UNARY_EXPR) {|mult_expr,op,unary_expr| Mathreematics.new(mult_expr, op,
unary_expr)}
        match(:UNARY_EXPR)
      end

      rule :UNARY_EXPR do
        match('-',:MATH_PRIMARY) {|_,math_primary| -math_primary}
        match('+',:MATH_PRIMARY) {|_,math_primary| math_primary}
        match('(',:MATH_EXPR, ')') {|_,math_expr,_| math_expr}
        match(:MATH_PRIMARY)
      end

      rule :MATH_PRIMARY do
        match(:SUBSCRIPTION)
        match(:FUNC_CALL)
        match(:IDENTIFIER)
        match(:NUM_LIT)
        match(:BOOL_LIT)
      end
    end
  end


  ### ###################### ###
  ### <3 "MAIN"-FUNKTION 3>  ###
  ### ###################### ###

  # När three anropas körs parsningen och evalueringen #

  def three
    puts("\nWelcome to <3!\nPlease specify the source code file to be interpreted (ex: 3.three):")
    source =  gets.chomp

    file = File.read(source)
    puts("\n--- Executing: #{source} --- \n\n")
```

```
    @threeParser.parse(file)
  end

  # Debug-utskrifter till parsern
  def log(state = false)
    if state
      @threeParser.logger.level = Logger::DEBUG
    else
      @threeParser.logger.level = Logger::WARN
    end
  end
end


################################################
#            <3 MAIN SEQUENCE 3>               #
################################################

t = LessThanThree.new
t.log

# Kör!
begin
  t.three
rescue Exception => error
  puts error.message
end

puts("\n-----------------------------------------")
puts("Reached end of program. Three you later! <3 \n")


################################################
#     THREE YOU LATER! <3++ COMING SOON        #
################################################
```

Bilaga 4. 3_tree.rb

```
#!/usr/bin/env ruby -w
# -*- coding: iso-8859-1 -*-

############################ 3_tree.rb ######################################
#                                                                           #
# <3 by Hannah Börjesson and Per Jonsson 2013                               #
# @Innovativ Programmering, Linköping university                            #
#                                                                           #
# A lovable interpreted imperative programming language                     #
#                                                                           #
# This file contains a syntax tree builder with eval methods for each node  #
#                                                                           #
# The rest will follow in Swedish                                           #
#                                                                           #
#############################################################################

=begin
Följande Scope-klass är inspirerat av en liknande klass i språket "Nibla"
(Albin Ekberg), som är ett av projekten från 2012 års upplaga av TDP019
=end
class Scope
  attr_accessor(:parent)
  def initialize
    @@counter = 1
    @@scope =  {}
  end
  def reset
    set(s)
  end
  def revert
  end
  def Scope.set(s)
    @@scope = s
  end
  def Scope.get
     @@scope
  end
  def Scope.counter
```

```
    @@counter
  end
  def Scope.create
    scope = {}
    @@counter += 1
    scope[@@counter] = Scope.get
    scope
  end
  def Scope.reset(s)
     Scope.set(s[@@counter])
    @@counter -= 1
  end
end

############################
Scope.new ## Skapa scope ##
############################

# Den översta noden vars eval-funktion anropas först
class Program
  attr_accessor
  def initialize(statements)
    @statements = statements
  end

  def eval
      @statements.eval
  end
end

class Statements
  def initialize(statement, *statements)
    @statements = statements
    @statement = statement
    @return = Null.new
  end

  def eval
    @statements << @statement
    @statement = Null.new
```

```
    # Måste göra reverse för att det ska bli top-down order
    @statements.reverse_each do |s|
      return s if s.is_a?(Return || Break)
      @return = s.eval
    end
   @return
  end
end

class Null
  def eval
    self
  end
end

class Break
  def eval
    self
  end
end


class Return
  def initialize (return_arg)
    @return_arg = return_arg
  end
  def eval
    @return_arg.eval
  end
end

class Assign
  def initialize(target, assignment)
    @target = target
    @assignment = assignment
  end

  def eval
    @scope = Scope.get
```

```
    value = @assignment.eval

    # Hämta ut variabelns värde från Scope genom att använda namnet som nyckel
    if @target.is_a?(Subscription)
      raise "<3 Assign Error: Target is not subscriptable" unless @scope[@target.name]
      @scope[@target.name][@target.index.eval] = value
    else
      @scope[@target.name] = value
    end

    value
  end
end

=begin
Följande Variabelklass med look-up-funktion är inspirerat av en liknande klass
 i språket "Nibla" (Albin Ekberg)som är ett av projekten från 2012 års TDP019
=end
class Variable
  attr_accessor(:name)

  def initialize(id)
    @name = id
  end

  def eval
    look_up(self)
  end
end

def look_up(variable)
  if variable.is_a?(Variable) then name = variable.name  else name = variable end

  scope = Scope.get

  # Sök i scope om variabeln finns i närmaste omgivningen
  if scope.has_key?(name)
    return scope[name]
  # Annars sök i en underliggande omgivning
  elsif scope[Scope.counter] != nil
```

```
    counter = Scope.counter

    while counter != 1
      if scope[counter].has_key?(name)
        return scope[counter][name]
      else
        # Återskapa underliggande omgivning
        scope = scope.values[0] # första entryn är ett scope
        scope = Hash[*scope.collect {|x| [x]}.flatten]
        counter -= 1
      end
    end
  end

  raise "<3 Name Error: The variable #{name} does not exist"
end

class List < Array
  def initialize(values)
    self << values if values
  end

  def eval
     # Om values är nil, evaluera ej, returnera då tom List
    self.flatten!
    self.collect! {|s| s.eval } # Evaluera varje element
    self.reverse! # Rotera för att skapa rätt ordning
  end

  def subscript(index)
    if index.is_a?(Array) # indexering ser ut så här: [][], etc
      index.each do |i|
        @value_at_i = self[i.eval]
      end
    else # Indexering med endast en []
      @value_at_i = self[index]
    end
    @value_at_i # returnera värdet på positionen i, i listan
  end
end
```

```ruby
class Subscription
  attr_accessor(:name, :index)
  def initialize(container, index)
    @container = container
    @index = index
    @name = @container
  end
  def type
    @container.class
  end

  # Måste kolla så att identifierns variabel är subscriptable, List eller string
  def eval
    container = look_up(@container)
    container.subscript(@index.eval) # Returnera resultatet
  end
end

class SubValues
  def initialize(*digits)
    @digits = digits
  end
  def eval
    @digits
  end
end

class Function
  def initialize(func_name, statements, *pars) # pars = nil
    @func_name = func_name
    @statements = statements
    @pars = pars
  end

  def eval
    # Lokalisera omgivning att spara funktionen i
    @scope = Scope.get
    # Spara parametrar och satser (funktionskropp)
    func_body = Hash.new
```

```
    func_body[:pars] = @pars
    func_body[:statements] = @statements

    # Kolla om vi ska skriva över funktionen
    counter = Scope.counter
    while counter > 1
      if @scope[counter].has_key?(@func_name)
        @scope[counter][@func_name] = func_body
        return
      else
        @scope = @scope.values[0]
        @scope = Hash[*@scope.collect {|x| [x]}.flatten]
        counter -= 1
      end
    end

    # Vi lägger i funktionen i scope-tabellen om den inte finns
    @scope = Scope.get
    @scope[@func_name] = func_body
  end
end


class ParameterList < Array
  def initialize(*parameters)
    self << parameters
  end

  def eval
    self.each do |p|
      p = p.eval
    end
    self
  end
end

class Parameter < Hash
  def initialize(var_name, value = nil) # value är valfritt
    # Tilldela parametern en egen variabeltabell
    (value) ? self[var_name] = value.eval : self[var_name] = value
```

```
  end
  def eval
    self
  end
end

class FunctionCall
  def initialize(func_name, args = nil) # args valfritt
    @func_name = func_name
    @args = args
  end

  def eval
    # Om funktionen anropas med ett värde, eller en Variable med ett värde:
    # kör look_up om det finns en Variable (den måste finnas i scope redan)
    # Vi skriver då över argumenten från en Variabel till ett värde
    if @args
      0.upto(@args.flatten!.length) do |i|
        @args[i] = look_up(@args[i]) if @args[i].is_a?(Variable)
      end
    end

    # Hämta funktionskropp: statements och parametrar (lokalisera i scope)
    func_body = @func_name.eval

    # Spara undan statements separat
    statements = func_body[:statements]

    # Skapa ett lokalt scope för att evaluera funktionens
    # satser och inrymma lokala variabler och värden
    @scope = Scope.create
    Scope.set(@scope)

    pars = {} # Lokal parameterlista
    # Om det finns parametrar, evaluera dessa
    if func_body[:pars]
      # func_body[:pars] lagrar en array med parametrar
      # itererera genom varje parameter, evaluera och spara
      func_body[:pars].each do |par|
        if par.is_a?(ParameterList)
```

```
      par.flatten.each do |p|
        # uppdatera vår Hash med en enskild post åt gången
        pars.merge!(p.eval)
      end
    else # Endast en parameter
      pars = par.eval # Spara som parameterlista
    end
  end
end


# Eventuellt skriva över motsvarande argument i Scope
(@args) ? args_len = @args.length : args_len = 0

# Matcha antal argument mot parameterlistan (ej för många, ej för få)
# args.length får inte understiga antalet nilparametrar,
# eller överstiga totala antalet parametrar
# om det är korrekt, kan vi skriva över värdena
nil_pars=0
pars.each_value {|v| nil_pars+=1 unless v }

arg_error = "<3 Argument Error: Wrong number of arguments; (#{args_len} args, expected #{nil_pars}
args)"
raise arg_error if args_len < nil_pars || args_len > pars.length

if @args
  # För varje argument som kommer in, skriv över motsvarande parameter
  pars.each_key do |p|
    pars[p] = @args.shift unless @args.empty?
  end
end

# Vi behöver se till att parametrarna får sina rätta värden
# (överskrivna eller default)
# innan vi evaluerar satserna i funktionen.
# Det gör vi med hjälp av Assign-konstruktionen
assign_statements = [];

# Spara färdiga parametrar i lokalt scope och lägg till eventuella värden
pars.each do |par_name, value|
  assign_statements << Assign.new(par_name, value.eval)
```

```
    end

    # Evaluera satser, efter att evaluering av parametrar/argument gjorts
    # Så att de har en omgivning att evalueras utifrån
    # Se till att vi evaluerar i rätt ordning
    assign_statements.each {|a| a.eval } # Se till så att vi inte skriver över globalt

    @return = statements.eval
    @return = @return.eval # Returvärde måste evalueras igen

    # Återställ omgivning
    Scope.reset(@scope)
    # Returnera returvärde
    @return
  end
end

class ArgumentList < Array
  def initialize(*args)
    self << args
  end
  def eval
    self.each do |a|
      p = a.eval
    end
    self
  end
end

class String
  def name
    self
  end
  def eval
    self
  end
end

class Integer
  def eval
```

```
    self
  end
end

class Float
  def eval
    self
  end
end

class TrueClass
  def eval
    true
  end
end

# Hanterar både Float och Integer
class NumLit
  def initialize(value)
    @value = value
  end
  def eval
    @value
  end
end

class BoolLit
  def initialize (value)
    @value = value
  end
  def eval
    @value
  end
end

class Expression
  def initialize(lhs, op, rhs = nil) # nil om not-test
    @lhs = lhs
    @op = op
    @rhs = rhs
```

```
  end

  def eval
    if not @rhs
      return (not @lhs.eval)
    else
      Kernel.eval("#{@lhs.eval} #{@op} #{@rhs.eval}")
    end
  end
end

class Comparison
  def initialize(lhs, op=nil, rhs=nil)
    @lhs = lhs
    @op = op
    @rhs = rhs
  end

  def eval
    if @lhs.is_a?(BoolLit)
      @lhs.eval
    else
      Kernel.eval("#{@lhs.eval} #{@op} #{@rhs.eval}")
    end
  end
end

class Mathreematics
  def initialize(lhs, op, rhs)
    @op = op
    @lhs = lhs
    @rhs = rhs
  end

  def eval
     Kernel.eval("#{@lhs.eval} #{@op} #{@rhs.eval}")
  end
end

class IfStmt < Array
```

```
  # En if-del måste alltid förekomma, dock ej de två andra grenarna
  def initialize(if_part, elseif_part = nil, else_part = nil)
    self << if_part
    self << elseif_part.flatten if elseif_part
    self << else_part if else_part
    self.flatten!
  end

  def eval
     self.each do |stmt|
      # Om vi har ett stmt som inte är nil:
      # evaluera, fånga returvärde, avbryt if-blocket om returvärdet är falskt
      # för då har vi evaluerat en gren och kan stanna där
      if stmt then
        @return = stmt.eval
        break unless @return == true # vi har gått igenom en If
        return @return.eval if @return.is_a?(Break || Return) # break:a/return:a
      end
    end
    @return
  end
end

class ElseIfs < Array
  def initialize(*parts)
    self << parts.flatten
  end
end

class If
  def initialize(statements, expression = nil)
    @expression = expression
    @statements = statements
  end

  def eval
    if @expression # expression ej nil
      if @expression.eval # Är uttrycket till (if|elseif) true ?
        @return = @statements.eval
        if @return.is_a?(Break || Return) then return @return end # d.v.s. break
```

```
        return @return # Gå inte vidare till andra grenar efter evaluering
      else # @expression == false
      end
    else # else-grenen (ej if/elseif => inget villkor (@expression))
      return @statements.eval
    end
    true
  end
end

class Loop
  def initialize(expression, statements)
    @statements = statements
    @expression = expression
  end

  def eval
    while @expression.eval
      @return = @statements.eval
      return Null.new if @return.is_a?(Break) # break:a
      return @return if @return.is_a?(Return) # returnera
     end
    # Returnera
    @return
  end
end

class Print
  def initialize(print_item)
    @p = print_item
  end

  def eval
    @p = @p.eval
    if @p.is_a?(String) # ta bort escape:ade citationstecken
      puts @p.gsub(/"/, '')
    else
      puts @p
    end
  end
```

```
end

class Read
  def initialize(msg)
    @msg = msg
  end

  def eval
    @msg = @msg.eval
    if @msg.is_a?(String) # ta bort escape:ade citationstecken
      puts @msg.gsub(/"/, '')
    else
      puts @msg
    end
    input = gets.chomp
    input
  end
end
```

Bilaga 5. 3_parser.rb

```ruby
#!/usr/bin/env ruby

# 2010-02-11 New version of this file for the 2010 instance of TDP007
#   which handles false return values during parsing, and has an easy way
#   of turning on and off debug messages.

require 'logger'

class Rule

  # A rule is created through the rule method of the Parser class, like this:
  #   rule :term do
  #     match(:term, '*', :dice) {|a, _, b| a * b }
  #     match(:term, '/', :dice) {|a, _, b| a / b }
  #     match(:dice)
  #   end

  Match = Struct.new :pattern, :block

  def initialize(name, parser)
    @logger = parser.logger
    # The name of the expressions this rule matches
    @name = name
    # We need the parser to recursively parse sub-expressions occurring
    # within the pattern of the match objects associated with this rule
    @parser = parser
    @matches = []
    # Left-recursive matches
    @lrmatches = []
  end

  # Add a matching expression to this rule, as in this example:
  #   match(:term, '*', :dice) {|a, _, b| a * b }
  # The arguments to 'match' describe the constituents of this expression.
  def match(*pattern, &block)
    match = Match.new(pattern, block)
    # If the pattern is left-recursive, then add it to the left-recursive set
```

```
  if pattern[0] == @name
    pattern.shift
    @lrmatches << match
  else
    @matches << match
  end
end

def parse
  # Try non-left-recursive matches first, to avoid infinite recursion
  match_result = try_matches(@matches)
  return nil if match_result.nil?
  loop do
    result = try_matches(@lrmatches, match_result)
    return match_result if result.nil?
    match_result = result
  end
end

private

# Try out all matching patterns of this rule
def try_matches(matches, pre_result = nil)
  match_result = nil
  # Begin at the current position in the input string of the parser
  start = @parser.pos
  matches.each do |match|
    # pre_result is a previously available result from evaluating expressions
    result = pre_result ? [pre_result] : []

    # We iterate through the parts of the pattern, which may be e.g.
    #   [:expr,'*',:term]
    match.pattern.each_with_index do |token,index|

      # If this "token" is a compound term, add the result of
      # parsing it to the "result" array
      if @parser.rules[token]
        result << @parser.rules[token].parse
        if result.last.nil?
          result = nil
```

```
            break
          end
         # @logger.debug("Matched '#{@name} = #{match.pattern[index..-1].inspect}'")
        else
          # Otherwise, we consume the token as part of applying this rule
          nt = @parser.expect(token)
          if nt
            result << nt
            if @lrmatches.include?(match.pattern) then
              pattern = [@name]+match.pattern
            else
              pattern = match.pattern
            end
            @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #{pattern.inspect}'")
          else
            result = nil
            break
          end
        end
      end
      if result
        if match.block
          match_result = match.block.call(*result)
        else
          match_result = result[0]
        end
        #@logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#{@name}' and generated
'#{match_result.inspect}'") unless match_result.nil?
        break
      else
        # If this rule did not match the current token list, move
        # back to the scan position of the last match
        @parser.pos = start
      end
    end

    return match_result
  end
end

class Parser
```

```
attr_accessor :pos
attr_reader :rules, :string, :logger

class ParseError < RuntimeError
end

def initialize(language_name, &block)
  @logger = Logger.new(STDOUT)
  @lex_tokens = []
  @rules = {}
  @start = nil
  @language_name = language_name
  instance_eval(&block)
end

# Tokenize the string into small pieces
def tokenize(string)
  @tokens = []
  @string = string.clone
  until string.empty?
    # Unless any of the valid tokens of our language are the prefix of
    # 'string', we fail with an exception
    raise ParseError, "unable to lex '#{string}" unless @lex_tokens.any? do |tok|
      match = tok.pattern.match(string)
      # The regular expression of a token has matched the beginning of 'string'
      if match
        @logger.debug("Token #{match[0]} consumed")
        # Also, evaluate this expression by using the block
        # associated with the token
        @tokens << tok.block.call(match.to_s) if tok.block
        # consume the match and proceed with the rest of the string
        string = match.post_match
        true
      else
        # this token pattern did not match, try the next
        false
      end # if
    end # raise
  end # until
```

```
  end

  def parse(string)
    # First, split the string according to the "token" instructions given.
    # Afterwards @tokens contains all tokens that are to be parsed.
    tokenize(string)

    # These variables are used to match if the total number of tokens
    # are consumed by the parser
    @pos = 0
    @max_pos = 0
    @expected = []
    # Parse (and evaluate) the tokens received
    result = @start.parse
    # If there are unparsed extra tokens, signal error
    if @pos != @tokens.size
      raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found
'#{@tokens[@max_pos]}'"
    end
    return result
  end

  def next_token
    @pos += 1
    return @tokens[@pos - 1]
  end

  # Return the next token in the queue
  def expect(tok)
    t = next_token
    if @pos - 1 > @max_pos
      @max_pos = @pos - 1
      @expected = []
    end
    return t if tok === t
    @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
    return nil
  end
```

```
  def to_s
    "Parser for #{@language_name}"
  end

  private

  LexToken = Struct.new(:pattern, :block)

  def token(pattern, &block)
    @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
  end

  def start(name, &block)
    rule(name, &block)
    @start = @rules[name]
  end

  def rule(name,&block)
    @current_rule = Rule.new(name, self)
    @rules[name] = @current_rule
    instance_eval(&block)
    @current_rule = nil
  end

  def match(*pattern, &block)
    @current_rule.send(:match, *pattern, &block)
  end

end
```