Obligatorisk uppgift: Symbolisk kalkylator

Moment: Dynamiska strukturer (särskilt trädstrukturer), klasshierarkier, arv, dynamisk bindning, polymorfi, undantag samt läsa och förstå kod.

Uppgiften består av att i första hand sätta sig in i ett existerande program och sedan modifiera och bygga ut det. (Enligt många före detta studenter är detta en av de vanligaste första arbetsuppgifterna ni kommer att drabbas av när ni kommer ut verkligheten)

Programmet blir betydligt större än de föregående programmen ni skrivit och det slutliga programmet kommer innehålla närmare 30 klasser. För att det skall gå att genomföra krävs det att du följer de instruktioner som vi ger i både skriftlig och muntlig form. Du kommer med största sannolikhet behöva flera genomläsningar av detta dokument för att förstå alla detaljer.

Uppgiftsbeskrivning

Du skall utveckla en kalkylator som, till skillnad från den numeriska kalkylatorn i uppgift 3, även kan utföra symboliska operationer som derivering.

I den numeriska kalkylatorn lästes och evaluerades uttryck direkt i parsern. I den här uppgiften ska ni i stället bygga upp en intern representation av det inlästa uttrycket och sedan evaluera den. Resultatet av en evaluering kan vara ett *symboliskt uttryck* och inte bara ett numeriskt värde.

För uttryck med numeriska värden skall det fungera som tidigare. Exempel:

```
Input  : 2*(-3-1+6)/0.5 + 0.5
Result : 8.5

Input  : log(exp(ans))
Result : 8.5

Input  : 1=x
Result : 1.0

Input  : (sin(x)*sin(x) = y) + cos(x)*cos(x)
Result : 1.0

Input  : y
Result : 0.7080734182735712

Input :  exp(2.0*x)+x=y
Result: 8.38905609893065
```

I språk som skall hantera uttryck behöver man ett explicit sätt att ange om man menar uttrycket som det står eller värdet av uttrycket. Antag att man i sista uttrycket i exemplet ovan vill att `y` skall få *uttrycket* `exp(2.0*x)+x` som värde, inte *värdet* av uttrycket (8.38905609893065). För detta för vi in en *kvot*-operator (`"`).

Exempel:

```
Input : "(exp(2*x) + x) = y
Result: exp(2.0*x)+x

Input : y
Result: exp(2.0*x)+x

Input : ans
Result: exp(2.0*x)+x

Input : "y
Result: y
```

Värdet av ett kvotat utttryck är alltså uttrycket själv.

Vi använder apostrofen (’) som binär operator för derivering (se syntax-diagrammen nedan).

Exempel:

```
Input : exp(x*x+1)’x
Result: 0.0

Input  : "exp(x*x +1)’x
Result : 2.0*x*exp(x*x+1.0)

Input  : "exp(x*x +1) = y
Result : exp(x*x+1.0)

Input  : y’x
Result : 2.0*x*exp(x*x+1.0)

Input  : "y’x
Result : 0.0
```

Observera att deriveringsoperatorn evaluerar det som står till vänster — därför blir resultatet av den första input-raden 0.

Om man hela tiden vill att t ex `x` skall betyda `x` och inte värdet av `x` kan man ge `x` det symboliska värdet `x`:

```
Input  : "x=x
Result : x

Input  : sin(3*x)’x
Result : 3.0*cos(3.0*x)

Input  : x*x + "a*x = y
Result : x*x+a*x

Input  : y
Result : x*x+a*x

Input  : y’x
Result : 2.0*x+a

Input  : y’x’x
Result : 2.0

Input  : exp(y’x’x - 1)
Result : 2.7182818284590455

Input  : exp((y’x’x - x’x)*x) + x
Result : exp(x)+x
```

I exemplet ovan har `y` fått värdet `x*x + a*x`. Hur skall vi göra om vi vill se vad `y` får för värde om vi ger `x` ett numeriskt värde? Vi behöver ett sätt att säga att "vi vill ha värdet av värdet av y". Det kan vi göra med *evaluerings*-operatorn (`&`). Denna unära operator *evaluerar sin evaluerade operand*.

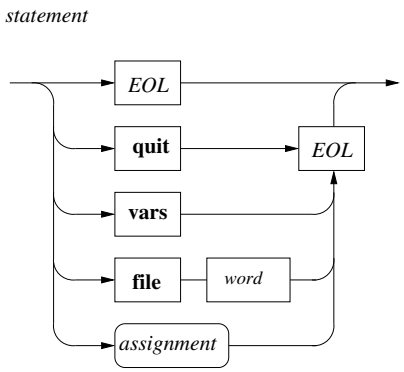
Exempel:

```
Input  : y
Result : x*x+a*x

Input  : &y
*** Evaluation error: Undefined varible: a

Input  : 2 = a
Result : 2.0

Input  : &y
Result : x*x+2.0*x

Input  : 3 = x
Result : 3.0

Input  : &y
Result : 15.0

Input  : "z = a
Result : z

Input  : &y
Result : 9.0+z*3.0

Input  : 100 = z
Result : 100.0

Input  : &y
Result : 9.0+z*3.0

Input  : &&y
Result : 309.0
```

Syntaxdiagram

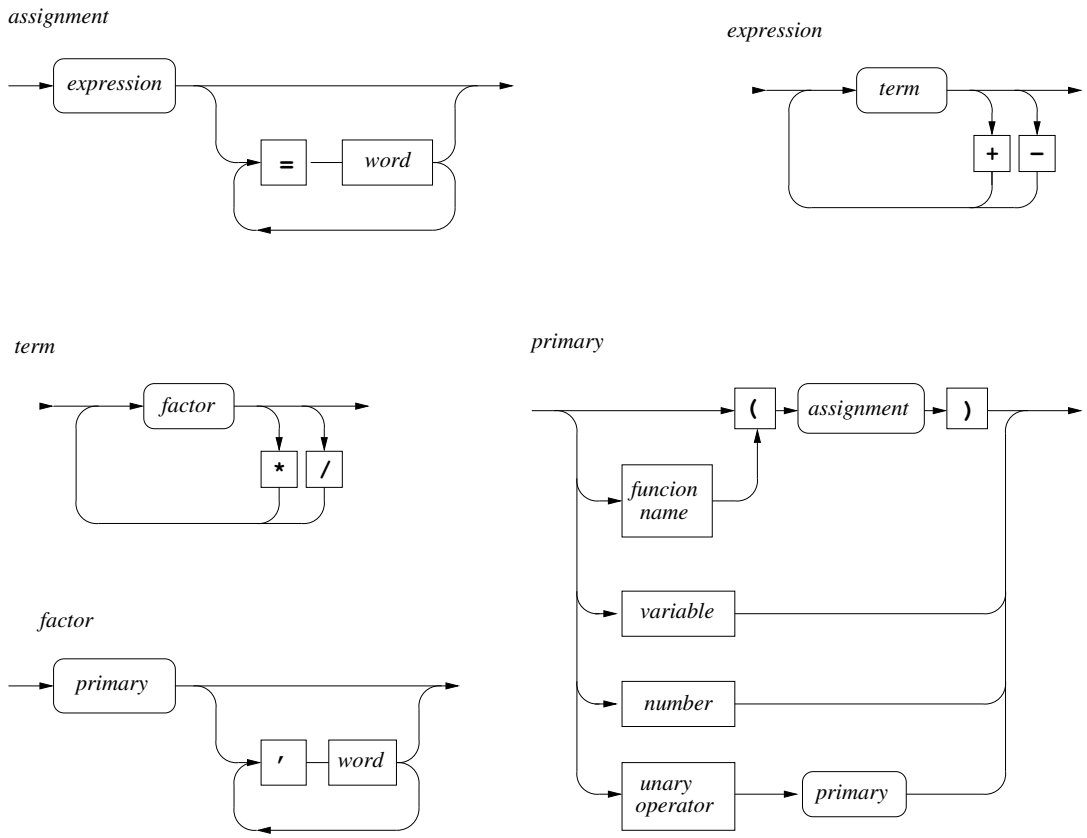
Syntaxen definieras av nedanstående diagram.

Klassen `Calculator` (som är helt given) hanterar satser med följande syntax:



Som synes hanteras ett nytt kommando `file` som används för att läsa indata från en fil. Kommandot `file test.txt` tar alltså input till kalkylatorn från filen `test.txt`.

Parsern använder väsentligen samma syntaxdefinition som tidigare. Metoden *factor* hanterar deriveringar och *primary* är kompletterad med eval- och kvotoperationerna.



I diagrammen står *function name* för `sin`, `cos`, `exp`, `log` och `abs` (absolutbelopp) samt *unary operator* för `-` (unärt minus), `"` (quote) och `&` (eval).

Programdesign

Intern representation av uttryck

Programmet skall läsa uttrycken och bygga upp en internrepresentation som lämpar sig för evaluering. Ni skall använda en *trädstruktur* där operatorer och funktioner lagras i de interna noderna och konstanter och variabler i löven.

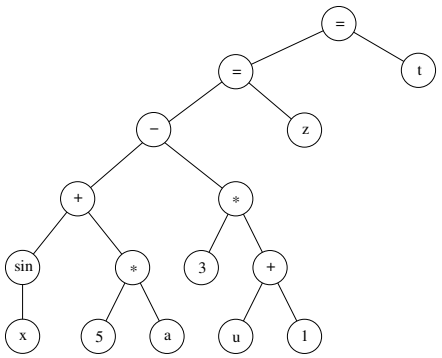
Det matematisk uttrycket

$$\sin x + 5a - 3(u + 1) = z = t$$

skall skrivas in som

```
sin(x) + 5*a - 3*(u+1) = z = t
```

och representeras av vidstående träd.



En utskrift av trädet (oevaluerat) blir

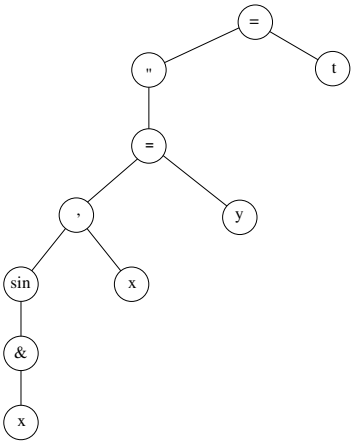
```
sin(x) + 5.0*a - 3.0*(u+1.0) = z = t
```

Naturligtvis skall även eval- kvot- och deriveringsoperatorerna representeras i trädet. Exempel:

Uttrycket

```
"(sin(&x)'x = y) = t
```

skall representeras som:



Gemensamt för alla noder är (bl a) att de skall kunna *evalueras* men evaluering görs olika beroende på vad det är för typ av nod (*konstant*, *variabel*, *addition*, ...).

Eftersom noderna ska kunna vara av olika typer (operatorer, funktioner, variabler, konstanter, ...) är det lämpligt att implementera dem i en klasshierarki med en generell basklass som skulle kunna kallas `SymbolicExpression`. Av layout-skäl kan det dock vara bra att använda ett kortare namn som `Sexpr` eftersom detta namn kommer att vara oerhört frekvent i koden.

Alla operationer (+, -, ...) och alla funktioner (*sin*, *cos*, ... skall ha egna klasser. Det skall också finnas en klass för variabler och en för konstanter.

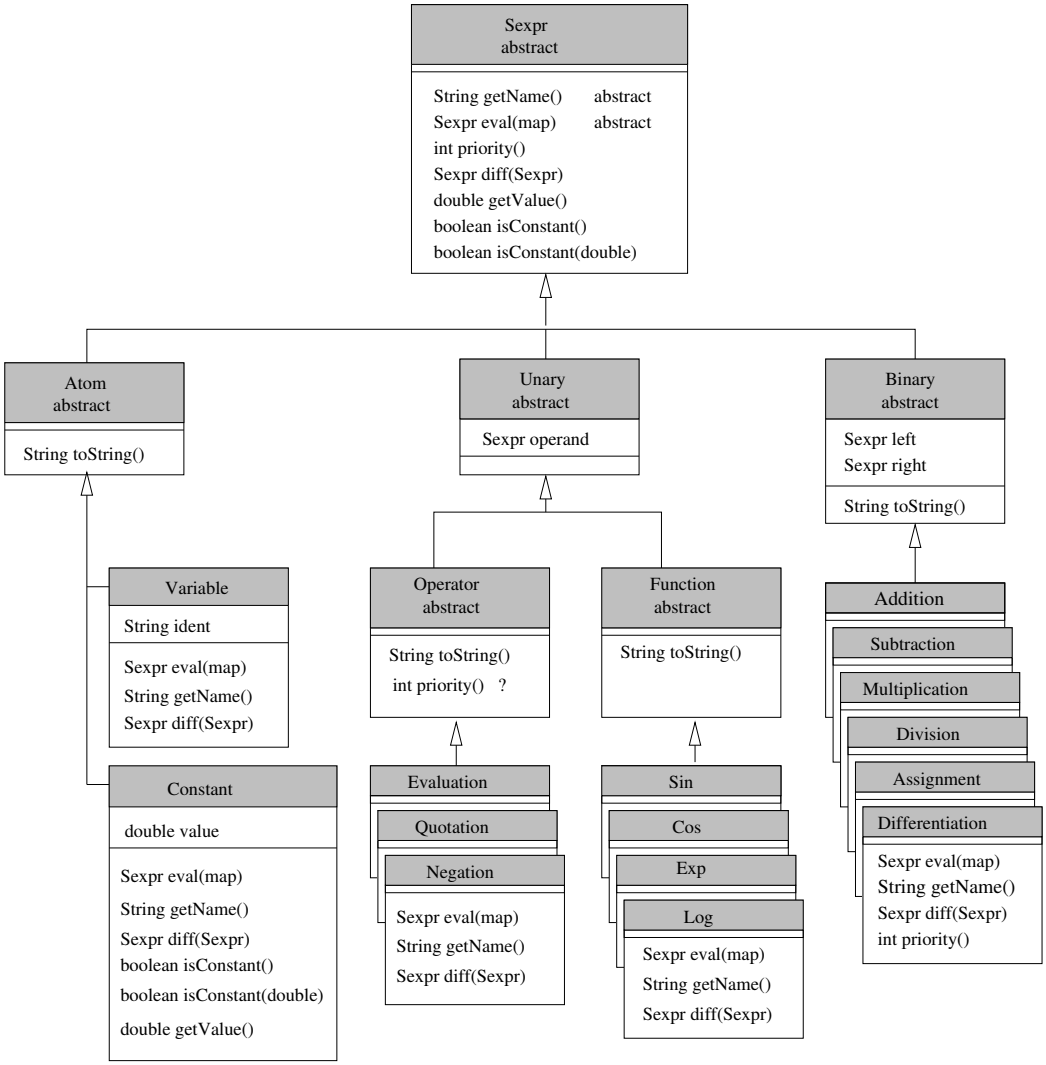
Det finns tre tydligt urskiljbara grupper av noder: de med *två* operander (*binära operatorer*), de med *en* en operand (*funktioner*, *unära operatorer*) och de utan operander (*konstanter*, *variabler*). Det är inte bara antalet operander (2, 1, 0) som är gemensamt för

varje grupp utan även vissa metoder. Sålunda kan bl a `toString`-metoderna skrivas på samma sätt t ex för alla binära operatorer.

Således skall man samla binära operatorer under klassen `Binary` och unära operatorer (inklusive funktioner) under klassen `Unary`. Eftersom funktioner och unära operatorer (unärt minus, kvot och eval) delvis skall hanteras annorlunda (olika `toString()`-metoder) så finns det två underklasser till `Unary`: `Function` och `Operator`.

Av "symmetriskäl" är det lämpligt att samla konstanter och variabler under klassen `Atom` även om det nog inte finns så mycket som kan utnyttjas gemensamt mellan dessa två grupper.

Det leder till följande design (ytterligare metoder behöver tillföras):



Obs 1: Konstruktorer är inte angivna.

Obs 2: Det är inte säkert att alla "lövklasser" ska implementera alla metoder. `Differentiation` och `Assignment` ska inte ha någon `diff`-metod (även om det råkar se ut så i diagrammet).

Basklassen `Sexpr` bör således ha följande utseende.

```
public abstract class Sexpr{
   ...
}
```

Ett exempel på en klass på mellannivån i hierarkin är `Unary`:

```
public abstract class Unary extends Sexpr {
   protected final Sexpr operand;
   public Unary(Sexpr operand){
      this.operand = operand;
   }

}
```

Hur bygger man internstrukturen?

I detta program är det parserns uppgift att bygga internstrukturen. Parserns metoder skall alltså inte returnera beräknade värden utan *referenser till en skapad nod.*

Parsern skall alltså ha samma struktur (samma metoder) som i den numeriska kalkylatorn *men* den

- skall *inte* utföra några beräkningar – metoderna skall i stället returnera referenser till de de skapade noderna,
- skall *inte* känna till mappen med variabelvärden och
- skall upptäcka syntaxfel som tidigare men inte beräkningsfel. (Den känner ju inte till några variabelvärden)

Ett anrop till `statement` skall alltså returnera en trädrepresentation av hela det uttryck som skrivits på raden.

Exempel: Parsermetoden `expression` får följande uteseende:

```
public Sexpr expression() {
    Sexpr sum = term();
    int c;
    while ((c=tokenizer.getChar())=='+' ||
           c == '-') {
       tokenizer.nextToken();
       if (c=='+') {
              sum = new Addition(sum, term());
       } else {
             sum = new Subtraction(sum, term());
       }
       return sum;
  }
```

Hur gör programmet för att skriva ut uttryck?

Som vanligt använder man `toString()`-metoder för att skapa en textrepresentation av de lagrade uttrycken. Dessa metoder gör i princip en inordertraversering av trädet. Ett problem är att i trädet är evalueringsordningen given av strukturen men i den externa bestäms den av prioriteter och parenteser. Således måste `toString`-metoderna, vid behov, sätta ut parenteser för att uttrycken skall bli korrekta. Initialt kan man göra det lätt för sig genom att *alltid* sätta ut parenteser men det blir inte så vackert.

Eftersom parenteshanteringen kan göras på samma sätt för alla unära respektive alla binära operatorer så placeras `toString`-metoderna i klasserna `Unary` respektive `Binary`. Själva operatornamnet erhålles av `getName()`

Så här ser klassen `Operator` ut med en ”simpleminded” (dvs en som sätter ut parenteser var sig det behövs eller ej) version av `toString`:

```
public abstract class Operator extends Unary {

   public String toString() {
      return getName() + "(" + operand.toString() + ")";
   }
   ...
}
```

Variabelvärden

För att hålla reda på variablers värden skall en `TreeMap` användas som tidigare. (Varför lägger man inte värdet som en egenskap hos variabeln?) Eftersom variablerna identifieras av sitt namn och värden är symboliska blir deklarationen:

```
Map<String,Sexpr> variables = new TreeMap<String,Sexpr>();
```

Variabellistan skall användas av *evalueringsmetoderna* (inte av parsern)!

Klassen `Calculator`

Klassen `Calculator` är given. Se

`http://www.it.uu.se/edu/course/homepage/prog2/ht15/assignments/symCalc/code/allFiles/Calculator.java`

Den är mycket lik motsvarande klass i uppgift 3. Den centrala delen hanterar alltså objekt av typen `Sexpr` i stället för typen `double` som i uppgift 3:

```
Sexpr parsed = parser.assignment();
System.out.println("Parsed: " + parsed);    // For debugging purposes
Sexpr value = parsed.eval(variables);
variables.put("ans", value);
System.out.println("Result: " + value.toString());
```

Till skillnad från körexemplen i detta dokument skrivs alltså även resultatet av parsningen ut vilket underlättar både vid felsökning och vid redovisningen.

Symbolisk evaluering

Som synes ovan evalueras ett lagrat uttryck genom att `eval`-metoden för dess rotnod anropas. Beroende på rotnodens typ kan eventuella operanders `eval`-metoder anropas. Dessa metoder utför helt enkelt en *postordertraversering* av trädet. En nods `eval`-metod avgör först vad som skall göras med operanderna. De flesta operatorers operander skall evalueras. Några undantag finns dock: tilldelningsoperatorn och deriveringsoperatorn som båda skall evaluera sin vänstra men inte sin högra operand (`1+2 = a` tilldelar *värdet* av `1+2` dvs 3 till variabeln `a` — inte till värdet av variabel `a`) samt `Quote`-operatorn som inte evaluerar sin enda operand.

Till evalueringsmetodernas finns klassen `Symbolic` som har ett antal statiska metoder som utför olika symboliska operationer: symbolisk addition, subtraktion, ... Dessa metoder utför numeriska operationer när det går och gör förenklingar när det går men returnerar alltid ett uttryck av typ `Sexpr`

Exempel:

Sinusklassens evalueringsmetod ser typiskt ut så här:

```
public Sexpr eval(Map<String,Sexpr> variables){
    return Symbolic.sin(operand.eval(variables));
}
```

och metoden `sin` i klassen `Symbolic` har följande utseende:

```
public static Sexpr sin(Sexpr arg){
   if (arg.isConstant())
       return new Constant(Math.sin(arg.getValue()));
   else
       return new Sin(arg);
}
```

Observera att metoderna i `Symbolic` aldrig anropar någon `eval`-metod!

Liknande metoder skrivs för de övriga funktionerna samt för alla aritmetiska operatorer. Om det är möjligt att utföra en numerisk beräkning ska det ske, annars får man utföra operationen symboliskt genom att skapa en ny nod av lämplig typ. Observera dock att metoden alltid måste returnera ett `Sexpr`, även om den lyckats beräkna ett numeriskt värde.

De "icke-aritmetiska" operatorerna (`=`, `"` och `&`) behöver inte ha några metoder i klassen `Symbolic`.

Exemplet ovan visar även att det finns ett behov av metoder som `isConstant()` och `getValue()` i klassen `Sexpr`.

Använd **inte instanceof** *och inte heller typkastning utan utnyttja den dynamiska bindningen!*

När en variabel evalueras söker man efter den i variabellistan. Hittar man den där returnerar man dess värde, annars kastas ett undantag. Värdet som hämtas ur listan skall *inte* evalueras.

En konstant evalueras alltid till sig själv.

Att evaluera en tilldelning innebär att man evaluerar den vänstra operanden och lägger in dess värde i variabellistan som den högra operanden (variabeln). Det är sedan värdet av den evaluerade operanden som returneras.

Derivering

I parsern hanteras deriveringsoperationer av metoden `factor` som skapar deriveringsnoder (objekt av klassen `Differentiation`).

Inplaceringen i syntaxdiagrammen gör att deriveringsoperatorn får en högre prioritet än multiplikation och division men lägre än unära operatorer.

Deriveringsoperationen definieras ju olika för olika operatorer och funktioner. För att han-

tera det förser man (nästan) alla ”lövklasser” med en med `diff` som definierar hur derivering av just detta element utförs.

Deriveringsmetoden skall ha en parameter som anger vilken variabel man deriverar med avseende på. I deriveringsmetoden använder man de metoder för symboliska operationer som finns i klassen `Symbolic`.

Klassen `Sin`:s deriveringsmetod ser ut så här:

```
public Sexpr diff(Sexpr variable){
    return Symbolic.mul(operand.diff(variable), new Cos(operand));
}
```

Hur och var skall deriveringsmetoden definieras för den oderiverbara funktionen (`abs`) och de oderiverbara operatorerna (`=`, `"`, `&`, `'`)?

Användningen av derivering kommer att visa på nödvändigheten av förenkling. En begränsad sådan kan uppnås genom modifiering av metoderna för symboliska operationer i `Symbolic`. Förenkling då samtliga inblandade argument är konstanter sker redan, men även om endast den ena operanden till en binär operator är konstant kan det vara möjligt att förenkla. Ett sådant exempel är addition med noll, men vilka de övriga fallen är får ni lista ut själva.

Krav för att bli godkänd på uppgiften

1. Programmet *skall* fungera i enlighet med specifikationen! Allt innehåll på den testfilen `test.txt` skall hanteras korrekt.
2. Koden *skall* följa kodkonventionerna och även på andra sätta vara lättläslig (inga rader får vara längre än 80 tecken, inga metoder längre än en skärmsida (med läslig font)).
3. Du *skall* använda den givna tokenizern (`Stokenizer`)
4. Du *skall* använda recursive descent som parse-metod.
5. Det *skall* finnas en metod för *varje* syntaktisk enhet (varje enskilt syntaxdiagram) och metoden skall ha de namn som anges där.
6. Du *skall* följa den givna klassdesignen i detalj dvs alla klasser skall finnas med de angivna namnen och de angivna metoderna.
7. Det är *förbjudet* att använda `instanceof` och typkastningar mellan olika klasser i klasshierarkin.
8. Upprepa inte identisk eller nästan identisk kod i olika klasser — varje klass skall bara definiera det som är speciellt för den. Tänk särskilt på detta i konstrukturer (utnyttja **super**!) och `toString`-metoder!

Arbetsgång

1. Kompilera och testkör den givna miniversionen och se vad som fungerar och vad som inte fungerar (bra). Läs koden och se till att du förstår hur den är uppbyggd.

2. Lägg till operationen ”subtraktion”. Du måste både ändra i parsern (se hur du gjorde i den numeriska kalkylatorn), skriva klassen `Subtraction` samt lägga till en metod i klassen `Symbolic`.
3. Skriv klassen `Differentiation` samt metoderna `diff(Sexpr)` i klasserna. Observera att deriveringsreglerna (`diff`-metoderna) skall använda sig av metoderna i klassen `Symbolic`. Förbättra metoderna i klassen `Symbolic` så att de utför triviala förenklingar (typ ”0*något”, ”1*något”, ”något - samma något” ...
4. Skriv de klasser som fattas i hierarkin.
5. Implementera `priority()`- och modifiera `toString()`-metoderna så att de bara sätter ut parenteser vid behov.
6. Använd filen `test.txt` som indata. (Filen fanns med blan de filer du laddade ner från början).
7. Redovisa!

Redovisningen

Vid redovisningen skall du

- Demonstrera att programmet fungerar med den senaste versionen av den givna testfilen och den den givna `Calculator`-klassen. Observera att dessa två kan uppdateras under kursens gång så kontrollera att du har rätt *innan* du försöker redovisa.
- Du skall kunna beskriva hur parsern går igenom ett uttryck och hur eval-metoderna fungerar.
- Du skall kunna besvara följande frågor:
    1. Varför är vissa metoder deklarerade som *abstrakta* i klassen `Sexpr`? Vad innebär det?
    2. Metoderna `priority()` och `isConstant()` är implementerade i klassen `Sexpr`. I vilka klasser behöver dessa omdefinieras?
    3. Instansvariablerna `left` och `right` i klassen `Binary` är deklarerade som `protected`. Vad innebär det?
    4. Objektet `variables` är deklarerad som en `Map` men instansierad till en `TreeMap`. Varför är den inte instansierad till en `Map`?
    5. Objektet `variables` skickas till alla `eval`-metoder men bara två använder den. Vilka? Varför skickas den ändå till alla?
    6. Vad behöver göras för att lägga till en ny funktion (t ex $\tan(x)$)