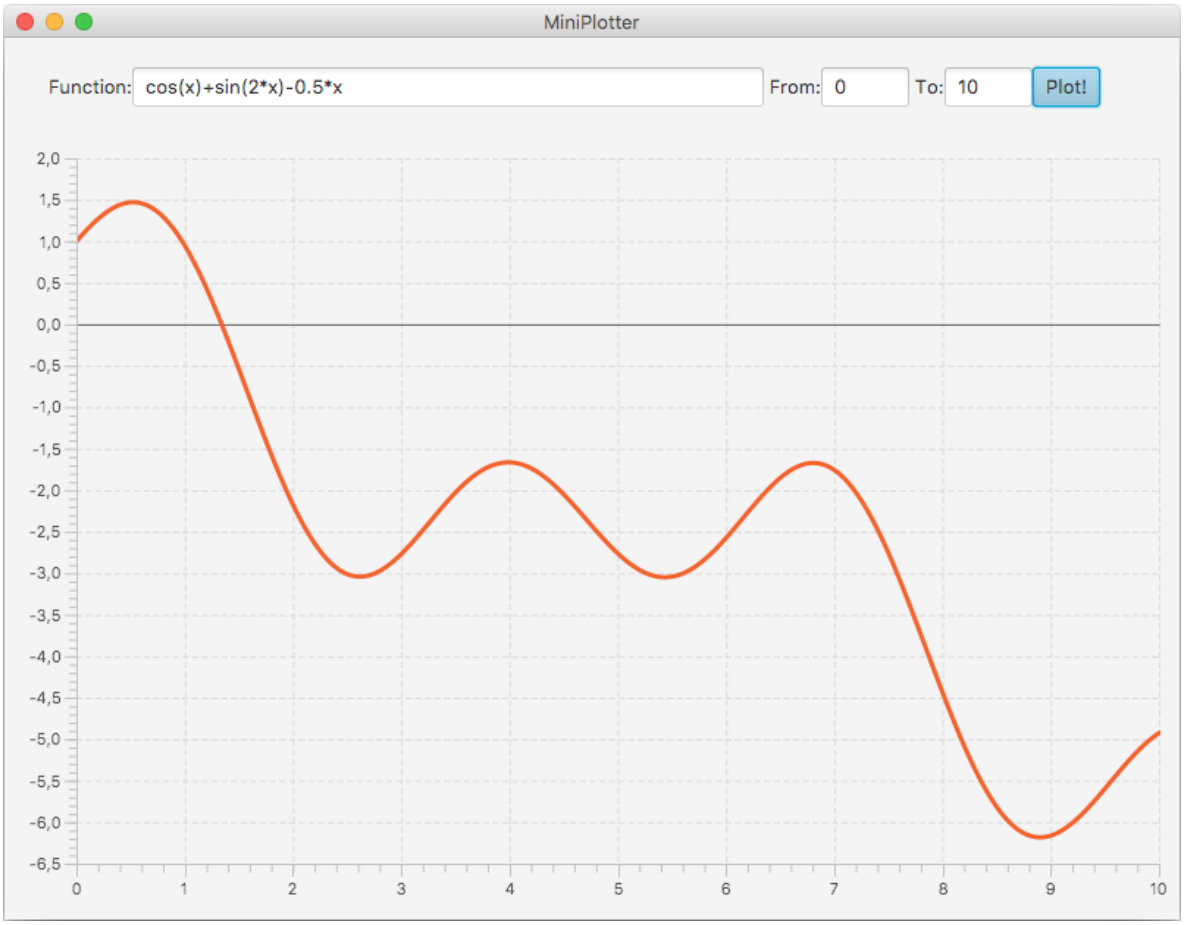
Inlämningsuppgift – MiniPlotter

I den här uppgiften ska ett program som ritar grafer av matematiska funktioner implementeras. Användaren kan mata in en funktion som text som sedan ritas upp inom ett visst intervall, vilket illustreras i Figur 1. Uppgiften ger träning i rekursion och träd samt en introduktion till hur man implementerar kompilatorer.



Figur 1: MiniPlotter

1 Problemet

Uppgiften kan delas in i tre delproblem:

1. Läsa in funktioner som text och skapa en Java-representation i form av ett aritmetiskt uttryckträd.
2. Beräkna funktionsvärdet (givet ett argument) för ett uttryckträd.
3. Implementera ett grafiskt användargränssnitt.

För att kunna lösa Delproblem 1 ovan behöver man förstå *lexikalisk* och *syntaktisk analys*.

2 Lexikalisk och syntaktisk analys

Processen att gå från en sträng till ett träd brukar delas in i två steg: *lexikalisk analys* och *syntaktisk analys*. Det första steget innebär att man går från en sträng till en följd av symboler (*tokens*). Det andra steget innebär att man går från en följd av symboler till ett träd. Den lexikaliska analysen anger vilka symboler som är tillåtna och den syntaktiska analysen anger i vilken ordning följden av symboler får komma i.

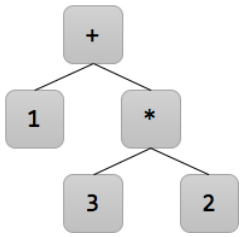
Exempelvis, om vi börjar med följande sträng (eller *mening*):

```
"1 + 3 * 2"
```

så kommer vi efter den lexikaliska analysen få följande följd av symboler:

```
"1" "+" "3" "*" "2"
```

I den lexikaliska analysen brukar man ta bort mellanslag som inte påverkar innebörden av meningen, som man kan se ovan. Analysen ser också till att `"11"` blir en symbol och `"1 1"` (med mellanslag i mellan) blir två symboler. Hur den lexikaliska analysen går till kommer vi inte att behandla i den här uppgiften, utan kommer att finnas färdigimplementerad. Därefter kommer vi i den syntaktiska analysen tolka symbolerna och skapa ett träd (om vi kan), som man kan se nedan:



Här kan man se att additionsoperatorn är roten till trädet och multiplikationsoperatorn är det högra barnet (eller den högra *operanden*) till additionsoperatorn. Att trädet är byggt på det här viset beror på att multiplikation har högre prioritet än addition, dvs, det matematiska uttrycket $1 + 3 * 2$ ska tolkas som $1 + (2 * 3)$ och inte som $(1 + 2) * 3$. Givet trädet ovan kan vi beräkna värdet på uttrycket genom att först multiplicera 3 med 2, vilket blir 6, och därefter addera 1 med resultatet 6 som blir 7. Beräkningen sker således som en postorder-traversering av trädet.

Den syntaktiska analysen kontrollerar också att meningen är syntaktiskt korrekt och bygger enbart ett träd om så är fallet. Ett exempel på en syntaktiskt felaktig mening är följande: `"1" "+" "*" "2"`. Vilka meningar som anses vara syntaktiskt korrekta beskrivs av en *kontextfri grammatik*.

2.1 Kontextfri grammatik

En kontextfri grammatik beskriver formellt vilka meningar (strängar) som syntaktiskt ingår i språket. Nedan följer en kontextfri grammatik (i EBNF-notation[1]) för ett enkelt aritmetiskt uttryckspråk L_{Add} bestående av addition och tal.

```
Expr -> Term ("+" Term)*
Term -> NUMBER
```

En kontextfri grammatik består av ett antal *produktioner*, där varje produktion har ett vänsterled och ett högerled. Vänsterledet består av en *icke-terminal* som produktionen ger en definition för. Högerledet består av en följd av icke-terminaler och *terminaler*. Terminaler motsvarar de symboler som man får från den lexikaliska analysen. Grammatiken för L_{Add} har två produktioner, två

[1] EBNF - Extended Backus-Naur Form - en vanlig notation för kontextfria grammatiker.

icke-terminaler, `Expr` och `Term`, samt två terminaler, `"+"` och `NUMBER`, där `NUMBER` är alla möjliga tal. I grammatiken ovan används också Kleene-stjärnan (`*`) som innebär repetition (noll eller flera). Detta medför att vi kan skriva godtyckligt långa uttryck av addition.

För att avgöra om en mening tillhör språket L_{Add} eller inte kan man försöka göra en *härledning*. Detta genom att applicera en sekvens av produktioner för att nå meningen från grammatikens *startsymbol*. Startsymbolen för L_{Add} är `Expr`. I varje steg i härledningen ersätts en icke-terminal med en definition av den. Om vi exempelvis har följden `"5" "+" "2"` kan vi göra följande härledning där vi börjar med startsymbolen:

```
Expr                 (ersätt Expr med högerledet i första produktionen)
=> Term "+" Term     (ersätt första Term med högerledet i andra produktionen)
=> NUMBER "+" Term
=> NUMBER "+" NUMBER
```

Således har vi bevisat att meningen `"5+2"` är syntaktiskt korrekt och ingår i språket L_{Add}. Språket L_{Add} definieras som mängden av alla meningar som kan härledas från grammatikens startsymbol. I det här fallet är den mängden oändligt stor på grund av att man kan ha noll eller flera additioner. Försöker vi göra en härledning av följden `"5" "+" "+" "2"` kommer vi att misslyckas eftersom grammatiken vill ha en `Term` mellan varje addition.

2.2 Recursive descent-parser

Ett sätt att implementera syntaktisk analys är att skriva en *recursive descent-parser*. Varje produktion blir då en metod med icke-terminalen i vänsterledet som metodnamn och högerledet översätts till metodkroppen. Varje icke-terminal i högerledet blir då ett metodanrop.

En implementation av grammatiken för språket L_{Add} som enbart anger om en mening är syntaktiskt korrekt eller ej illustreras i nedanstående kod.

```
public class Parser {
    private Iterator<String> remainingTokens;
    private String token;

    public void parse(List<String> tokens) throws ParseException {
        remainingTokens = tokens.iterator();
        token = remainingTokens.next();
        expr();
    }

    private void expr() throws ParseException {
        term();
        while (token.equals("+")) {
            accept();
            term();
        }
    }

    private void term() throws ParseException {
        if (isNumber()) {
            accept();
        } else {
            throw new ParseException("Unexpected token: " + token);
        }
    }

    ...
}
```

Klassen `Parser` läser en följd av symboler (tokens). Klassen används genom ett anrop till den publika metoden `parse(...)` som tar följden av symboler som parameter. Metoden `parse` kastar ett undantag av typen `ParseException` om följden av symboler är syntaktiskt felaktig. Metoden börjar med att skapa en iterator av symbolerna och sparar därefter första symbolen i attributet `token`. Attributet `token` håller hela tiden reda på den nuvarande symbolen som tolkas. Därefter anropas startsymbolens metod, dvs, metoden `expr()`.

I metoden `expr` kräver vi först en term, därför anropar vi metoden `term`. Därefter kan det komma en eller flera additioner som också kräver en term. Metoden `accept` anger att den nuvarande symbol är tillåten så här långt och vi flyttar fram ett steg, dvs, attributet `token` ändras till nästa symbol i följden.

I metoden `term` kontrollerar vi om den nuvarande symbolen är ett tal, vilket är korrekt, annars har vi ett syntaktiskt fel och kastar ett undantag.

Vi kan nu testa klassen genom att skapa en följd av symboler och sedan anropa metoden `parse`:

```
List<String> tokens = new ArrayList<String>();
tokens.add("5");
tokens.add("+");
tokens.add("2");
try {
    Parser p = new Parser();
    p.parse(tokens);
    System.out.println("Valid syntax!");
} catch (ParseException e) {
    System.out.println("Parse error!");
}
```

Om ett undantag kastas så är meningen syntaktiskt felaktig, annars är den syntaktiskt korrekt.

2.3 Slutet av meningen

I parsern behöver vi också kontrollera att vi har nått slutet av meningen, så att det inte finns några efterföljande symboler som inte har tolkats av parsern (exempelvis meningen `"1+2 35"`, där `"35"` borde ge syntaxfel). Detta kan man göra genom en liten ändring i slutet av `parse`-metoden:

```
public void parse(List<String> tokens) throws ParseException {
    remainingTokens = tokens.iterator();
    token = remainingTokens.next();
    expr();
    if (token != EOF_TOKEN) {
        throw new ParseException("Cannot parse: " + token);
    }
}
```

Här tvingar vi parsern att nå slutet av följden av symboler efter att vi har läst in icke-terminalen `Expr`. Strängen `EOF_TOKEN` är en särskild sträng som markerar att det inte finns några fler symboler. Den sätts i metoden `accept` när man flyttar fram nuvarande symbol ett steg och det inte finns någon mer symbol att läsa.

3 Semantisk analys

Efter att man vet att det aritmetiska uttrycket är syntaktiskt korrekt så vill man beräkna *meningen* av det, dvs, *semantiken* av uttrycket. För att göra detta låter man parsern skapa ett *abstrakt*

syntaxträd (AST) som man kan göra beräkningar över. Strukturen för det abstrakta syntaxträdet beskrivs av vanliga Java-klasser. För språket L_{Add} kan man beskriva strukturen med följande klasser:

```
public abstract class Expr { ... }
public class AddExpr extends Expr {
    private final Expr left, right;
    ...
}
public class NumExpr extends Expr {
    private final double value;
    ...
}
```

För uttrycket `"1+2+3"` låter man parsern skapa följande AST:

```
new AddExpr(new AddExpr(new NumExpr(1), new NumExpr(2)), new NumExpr(3))
```

Notera att detta motsvarar $(1 + 2) + 3$, dvs, addition är vänsterassociativt (beräkningen sker från vänster till höger). För addition i matematikens värld spelar det ingen roll, däremot har det betydelse för subtraktion: $(1 - 2) - 3 \neq 1 - (2 - 3)$.

Med modifikationer i parsern kan man låta den skapa det abstrakta syntaxträdet. Varje metod får nu skapa AST-objekt och returnera dem. Metoderna `expr()` och `term()` ändras till följande:

```
private Expr expr() throws ParseException {
    Expr e = term();
    while (token.equals("+")) {
        accept();
        Expr e2 = term();
        e = new AddExpr(e, e2);
    }
    return e;
}
private Expr term() throws ParseException {
    Expr e = null;
    if (isNumber()) {
        e = new NumExpr(Double.valueOf(token));
        accept();
    } else {
        throw new ParseException("Unexpected token: " + token);
    }
    return e;
}
```

Som man kan se skapas `AddExpr`-objekt i `expr`-metoden och `NumExpr`-objekt i `term`-metoden. Metoden `expr()` kommer att bygga trädet vänsterassociativt.

3.1 Beräkning av uttryck

Nu när vi har ett abstrakt syntaxträd vill vi kunna beräkna värdet av det. Exempelvis ska uttrycket `"1+2+3"` beräknas till 6. För att kunna göra detta behöver vi lägga till en metod `value` i AST-klasserna, som följande:

```
public abstract class Expr {
    public abstract double value();
}
public class AddExpr extends Expr {
    ...
```

```
    public double value() {
        return left.value() + right.value();
    }
}
public class NumExpr extends Expr {
    ...
    public double value() {
        return value;
    }
}
```

Notera att vi behöver införa metoden i klassen `Expr` eftersom typen på attributen `left` och `right` i `AddExpr` är av typen `Expr`. Således måste metoden `value` finnas i `Expr`.

Vi kan nu anropa den nya metoden på objektet vi får tillbaka från parsern för att beräkna värdet på uttrycket:

```
Expr e = new Parser().parse(tokens);
System.out.println(e.value());
```

3.2 Särskild rotnod

När man implementerar semantisk analys är det ofta praktisk att ha en särskild rotnod (som till exempel inte är rekursivt definierad). För språket L_{Add} inför vi därför en särskild rotnod definierad av AST-klassen `Function`. Den består av ett uttryck och delegerar alla metoder till uttryckets metoder. Klassen `Function` definieras som följande:

```
public class Function {
    private Expr expr;
    ...
    public double value() {
        return expr.value();
    }
}
```

Detta medför en liten ändring i `parse`-metoden som helt enkelt skapar ett `Function`-objekt med uttrycket från `expr`-anropet som argument. Funktionsobjektet blir då returvärdet i metoden `parse`.

4 Uppgift

Uppgiften går ut på att stödja språket L som beskrivs av följande grammatik:

```
Expr -> Term ("+"|"-" Term)*
Term -> Factor ("*"|"/" Factor)*
Factor -> NUMBER | IDENTIFIER | "(" Expr ")" | IDENTIFIER "(" Expr ")"
```

I grammatiken används nu alternativ (`|`). Exempelvis kan `Factor` vara antingen ett tal, ett variabelnamn, ett uttryck i parentes eller ett funktionsanrop med ett argument. Variabelnamn är antingen `x` eller konstanten `pi`. Funktionsanrop är till någon av de fördefinierade funktionerna: `sin`, `cos`, `sqrt` (kvadratroten), `ln` (logaritmen med basen e) och `exp` (exponentialfunktionen med basen e).

Språket stödjer också operationerna addition, subtraktion, multiplikation och division. Som man kan se i grammatiken är addition och subtraktion i högerledet för `Expr` samt multiplikation och division i högerledet för `Term`. Detta beror på att muliplikation och division har högre

prioritet, dvs, uttrycket $1 + 2 * 3$ ska tolkas som $1 + (2 * 3)$. I en grammatik specifierar man alltså olika prioritet med olika nivåer av produktioner.

Till skillnad mot grammtiken för språket L_{Add} innehåller grammatiken för L rekursion. Detta på grund av att icke-terminalen `Expr` finns i högerledet till `Factor`. Således kan man nå `Expr` från `Expr`. Detta händer om man försöker göra en härledning av uttrycket `"(x)"` eller av uttrycket `"sin(cos(x)+2*x)"`. Att grammatiken är rekursiv medför att implementationen av parsern också blir rekursiv. Detta är dock inget problem om man översätter grammatiken enligt de regler som tidigare har beskrivits.

En annan klurighet i grammatiken gäller variabelnamn och funktionsanrop. Båda dessa två inleds med en identifierare. För att veta om det är ett variabelnamn eller funktionsanrop som man läst in behöver man kontrollera vilken symbol som kommer efter identifieraren. Om det är en vänsterparentes, då är det ett funktionsanrop, annars är det ett variabelnamn. Detta behöver man kontrollera i metoden för `Factor`.

I den här uppgiften finns redan den lexikaliska analysen färdigimplementerad (se klassen `TokenScanner`). En identifierare börjar med en bokstav och därefter följs av noll eller flera bokstäver eller siffror. Ett tal kan anges antingen med eller utan decimalpunkt. Detta kommer att kontrolleras i den lexikaliska analysen. I den syntaktiska analysen kan man särskilja på identifierare och tal genom att se på det första tecknet.

4.1 Deluppgifter

Syntaktisk analys. En recursive descent-parser ska implementeras för språket L. När man implementerar parsern är det bra att göra det i små steg. Utgå från parsern för språket L_{Add}. Gå sedan vidare med att lägga till *en* operation, exempelvis subtraktion, och testa så att den fungerar genom att lägga till testfall. Se på hur testfallen är skrivna för språket L_{Add}. Därefter fortsätt på samma sätt med nästa operation. I den syntaktiska analysen bör man inte kontrollera om rätt variabelnamn eller funktionsnamn används, utan det görs istället i den semantiska analysen.

Semantisk analys. I den semantiska analysen ingår det att beräkna värdet för ett uttryck och leta efter fel. Det behövs således två metoder: `value` och `collectErrors`. Metoden `value` har en parameter `x` av typen `double` där man kan ange värdet för variabeln x. Klassen som representerar funktionsanrop (ska ej förväxlas med `Function`-klassen) kommer att delegera sin beräkning till motsvarande metoder i Math-klassen i Javas standardbibliotek. Metoden `collectErrors` kommer att traversera trädet och leta upp om ett felaktigt variabelnamn används (exempelvis `y`) eller om ett felaktigt funktionsanrop används (exempelvis `coh`). Metoden har en lista av fel (strängar) som parameter och vid varje fel läggs en sträng till i listan. Listan skapas i metoden i `Function`-objektet.

Automatisk testning. Skriv testfall med JUnit för både den syntaktiska och semantiska analysen som är heltäckande för språket L.

Grafiskt användargränssnitt. Implementera ett grafiskt användargränssnitt likt det i Figur 1. Utöver de klasser som har gåtts igenom på föreläsningarna är klassen `LineChart`[2] användbar, där man kan ange en lista av punkter som en linje ritas emellan. I gränssnittet ska man kunna mata in en funktion som text och inom vilket intervall funktionen ska ritas ut. När funktionen ritas ut beräknar man värdet för funktionen för kanske 1000 värden på x och lägger in punkterna (x-värdet och det beräknade y-värdet) i grafen. I klassen `LineChart` kan man ange att bara visa linjerna och ej punkterna. Om funktionen har ett lexikaliskt, syntaktiskt eller semantiskt fel ska detta visas för användaren.

[2] `https://docs.oracle.com/javafx/2/charts/line-chart.htm`