Analys och representation av aritmetiska uttryck

När man vill skriva ett program som kan hantera aritmetiska uttryck behöver man en representation av uttryck som lämpar sig för de operationer man vill utföra. Om man bara vill läsa in och skriva ut uttryck går det utmärkt med klassen `String`. Vi kallar detta för den *konkreta* eller *externa* representationen. Om man vill kunna beräkna värden av eller förenkla uttryck behövs en representation som gör uttryckets struktur tydlig. Vi kallar det för *abstrakt* eller *intern* representation. När vi använder en abstrakt representation i ett program behöver vi metoder för att konvertera mellan abstrakt och konkret representation.

Konkret representation

Vi vill kunna hantera vanliga aritmetisk uttryck som 314, -1, $1+2*x+x*x$ $(x+1)*(x-1)$, $n-(n/d)*d$), etc. Uttryck består alltså av tal, variabler, de fyra vanliga aritmetiska operatorerna och parenteser. Av exemplen går det inte att avgöra om $-x$ är ett tillåtet uttryck. En mer formell beskrivning ger besked:

> Ett *uttryck* består av en eller flera *termer* separerade av enkla plus- eller minustecken. En *term* består i sin tur av en eller flera *faktorer* separerade av enkla multiplikations- eller divisions- tecken. En *faktor* är ett *tal*, en *variabel* eller ett *uttryck* inom parenteser. Ett *tal* består av en eller flera siffror och får inledas med ett minustecken. En *variabel* består av en eller flera bokstäver bland a–z.

Det finns en särskild formell notation, *Backus–Naur–form* (BNF), för detta:

```
expr ::= term (addop term)*
term ::= factor (mulop factor)*
factor ::= number | name | '(' expr ')'
addop ::= '+' | '-'
mulop ::= '*' | '/'
```

Varje rad namnger en *syntaxsymbol* och räknar upp beståndsdelarna. Syntaxsymbolen står till vänster om "::=" och beståndsdelarna till höger. Det som står inom parentes och följs av en asterisk får upprepas noll eller flera gånger. När det finns alternativa beståndsdelar separeras dessa av |-tecken som utläses *eller*. Det som omges av apostrofer står för sig själv. Vi ser alltså att en `addop` är ett plus- eller minus-tecken och att ett `expr` består av en eller flera `term`er separerade av plus- eller minus-tecken. Symbolen för alternativ binder svagast så att `'(' expr ')'` är ett av tre alternativ för `factor`.

Vi kan fortsätta på samma sätt med

```
number ::= unsignedNumber | '-' unsignedNumber
unsignedNumber ::=  naturalNumber |  naturalNumber '.'  naturalNumber
naturalNumber ::= digit (digit)*
digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
name ::= letter (letter)*
letter ::= 'a' | 'b' | ... | 'z'
```

De tre punkterna borde egentligen ersättas med alternativ för mellanliggande bokstäver.

Vi har nu en *grammatik* för aritmetska uttryck och kan göra en *härledning* för att visa hur en sträng kan konstrueras. En härldning börjar med grammatikens *startsymbol*, som enligt konvention brukar vara den symbol som finns i högerledet i den första raden. I varje härledningssteg ersätter man en syntaxsymbol med motsvarande högerled. Härledningen är klar när det inte finns några syntaxsymboler kvar. Några led i följande härledning av strängen `1+x` är överhoppade.

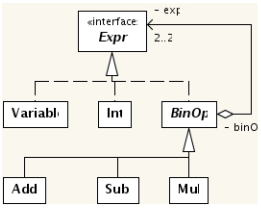
```
expr -> term addop term -> factor addop term -> number addop term ->
1 addop term -> 1 + term -> 1 + factor -> 1 + name -> 1 + x
```

Abstrakt representation

Man skulle kunna konstruera den abstrakta representationen utifrån grammatiken och representera ett uttryck som en lista med termer och en term som en lista av faktorer. Det finns dock ett litet problem med detta; vissa termer skall adderas och andra subtraheras och vissa faktorer skall multipliceras och andra ingå i divisioner. En enklare och bättre representation bygger på följande beskrivning av vad ett uttryck är.

> Ett uttryck är antingen ett tal, en variabel eller binärt. Ett binärt uttryck är antingen en addition, subtraktion, multiplikation eller division av två operander som båda är uttryck.

Denna beskrivning är mer abstrakt; den säger inget om parenteser och plustecken utan förutsätter att man vet vilken operation som skall representeras och vilka operanderna i ett binärt uttryck är. Den går direkt att översätta till ett klassdiagram.



Sammansatta uttryck måste representeras med objekt. En addition med två operander bör representeras som ett additionsobjekt med två attribut som båda är uttryck. Ett uttryck som bara består av ett tal skulle kunna representeras med en `double` men detta skulle då inte kunna användas i binärt uttryck. Alla utryck måste alltså representeras av objekt. Klassen `Double` duger inte heller för att representera ett uttryck som är ett tal eftersom den inte implementerar `Expr`.

Det är enkelt att implementera `toString` i de olika klassserna så att metoden returnerar en konkret representation av uttrycket. Om man bara vill se nödvändiga parenteser blir lite svårare.

Att göra en metod som implementerar inversen till `toString` är mer komplicerat. Metoden skall alltså ta en sträng som argument och returnera motsvarande abstrakta representation.

```
public Expr build(String string);
```

Syntaxanlys

Att kunna implementera en metod (*Factory method*-mönstret) ingår inte i kursen, men i inlämningsuppgift 2 får du använda en färdig sådan. Detta avsnitt är till för den som vill veta hur det går till. Kursen i kompilatorteknik innehåller mycket mer om detta problem.

Det är enkelt att skriva en metod som avgör om en sträng innehåller ett variabelnamn eller en följd av siffror och returnerar motsvarande `Variable`- eller `Int`-objekt. I kompilatortekniken kallas detta för en *lexikalanalysator*. I `java.io` finns en klass, `StreamTokenizer`, som gör jobbet. I följande kodavsnitt skapas ett sådant objekt som kommer att läsa från en sträng via en `StringReader`.

```
public class ExprParser extends StreamTokenizer {
    private int token;
    public ExprParser(String string) throws IOException  {
        super(new StringReader(string));
        ordinaryChar('-');
        ordinaryChar('/');
        token = nextToken();
    }
```

`StreamTokenizer` initieras så att minus- och divisionstecken behandlas som vanliga tecken. De är annars initierade för att passa analys av programspråk där man har negativa tal och kommentarer som inleds med `'/'`. `nextToken` returnerar ett heltal som anger vad för slags tecken den hittat.

I klassen finns metoder med samma namn som syntaxsymbolerna i grammatiken för uttryck. Metoden `expr` skall analysera ett helt uttryck enligt grammtiken

```
expr ::= term (addop term)*
```

Metoden `term` skall göra sammalunda med

```
term ::= factor (mulop factor)*
```

och `factor` skall klara av

```
factor ::= number | name | '(' expr ')'
```

Vi börjar med `factor`. Om nästa tecken (`token`) är en vänsterparentes så bygger vi ett helt uttryck genom att anropa `expr` rekursivt. Om `token` anger att den hittat ett tal `StreamTT_NUMBER` så finns detta att hämta i attributet `nval`.

```
    private Expr factor() throws IOException {
        Expr e;
        switch (token) {
            case '(' :
                token = nextToken();
                e = expr();
```

```
            token = nextToken();
            return e;
        case TT_NUMBER :
            double x = nval;
            token = nextToken();
            return new Int((int) x);
        case TT_WORD :
            String s = sval;
            token = nextToken();
            return new Variable(s);
    }
}
```

Om `token` i stället markerar att den hitta ett `StreamTT_WORD` finns strängen att hämta i `sval`.

Metoden `term` skall analysera en eller flera faktorer med hjälp av `factor`.

```
private Expr term() throws IOException {
    Expr result, factor;
    result = factor();
    while (token == '*' || token == '/') {
        int op = token;
        token = nextToken();
        factor = factor();
        switch (op) {
            case '*' :
                result = new Mul(result, factor);
                break;
            case '/' :
                result = new Div(result, factor);
                break;
        }
    }
    return result;
}
```

Slutligen skall `expr` ta hand om termerna. Detta är helt analogt med metoden `factor`. Detta lämnas som övning.

Slutligen definieras `build` som en synonym till `expr`.

```
public Expr build() throws IOException {
    return expr();
}
```