Syntaxanalys med ”recursive descent”

Rekursiva metoder är speciellt lämpliga när de data man skall behandla i sig är rekursivt uppbyggda. Ett exempel på detta är vanliga aritmetiska uttryck. Betrakta t.ex. följande uttryck

$$a + (b + c) \cdot d + e \cdot (f + g \cdot h)$$

Det finns ett antal regler för hur detta skall tolkas av typ ”multiplikation före addition”, ”parenteser först” och ”från vänster till höger vid lika prioritet”. Det är inte alldeles enkelt (men naturligtvis väl genomförbart) att realisera dessa regler i ett program som läser och tolkar ett uttryck. Vi kan emellertid definiera uttryck på följande sätt:

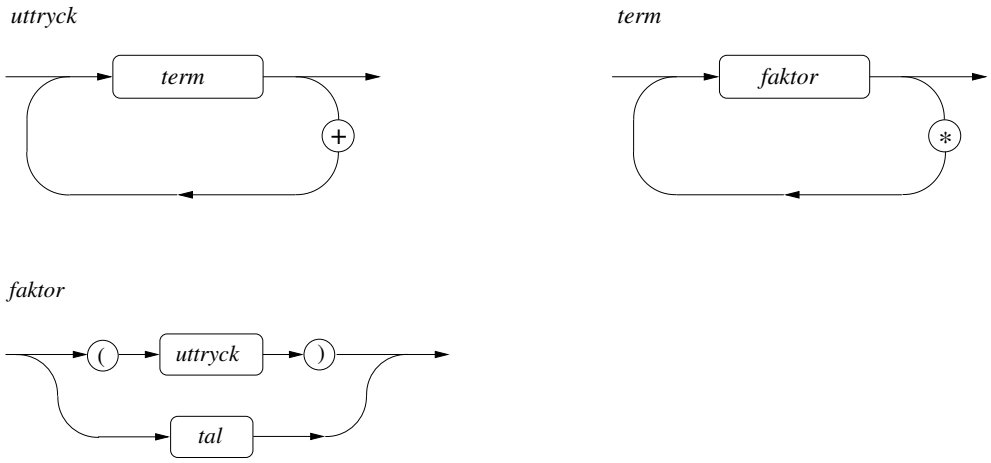
Ett *uttryck*
: är en sekvens av en eller flera *termer* med plustecken mellan.

En *term*
: är en sekvens av en eller flera *faktorer* med gångertecken mellan.

En *faktor*
: är antingen ett tal eller ett *uttryck* omgivet av parenteser.

(För enkelhetens skull begränsar vi oss till addition och multiplikation) Observera att ovanstående definition av uttryck är indirekt rekursiv.

Detta kan illustreras grafiskt med så kallade *syntaxdiagram*:



Vi har inte ritat någon figur för *tal* utan förutsätter att någon annan håller reda på hur ett tal definieras med hjälp av siffror, decimalpunkt mm.

Observera igen rekursionen: *uttryck* definieras med *term* som defineras med *faktor* som definieras med *uttryck*.

Alla figurerna innehåller ”vägskäl”. Det är det närmast kommande tecknet som avgör vilken väg vi skall gå. Om det t ex i *uttryck* kommer ett plustecken när en term har behandlats så skall vi gå tillbaka och ta en ny term. Om det *inte* kommer ett plustecken är uttrycket klart *oavsett vad som kommer* — det är någon annans problem att hantera det!

I *faktor* kan det hända ett det varken är en vänsterparentes eller ett tal som kommer. Då är det fel — indata uppfyller inte den föreskrivna syntaxen.

Med hjälp av syntaxdiagrammen är det enkelt att skriva ett program som korrekt hanterar generella former av denna begränsade variant av aritmetiska uttryck.

Vi skriver en metod som hanterar vart och ett av dessa begrepp (”syntaktiska enheter”). Vi låter också metoderna utföra den angivna operationen (additioner och multiplikationer):

```
double uttryck() {
  double sum = term();              // Först kommer en term
  while ( nextToRead() == '+' ) {   // därefter kan det komma ett plustecken
    readNextChar();                 // och sedan en term till som skall
    sum += term();                  // adderas till summan
  }
  return sum;
}

double term() {
  double prod = faktor();
  while ( nextToRead() == '*' ) {
    readNextChar();
    prod *= faktor();
  }
  return prod;
}

double faktor() {
   if ( nextToRead() == '(' ) {       // Om vänsterparentes
      readNextChar();                 // läs förbi '('
      double result = uttryck();
      if (nextToRead()==')') {        // kontrollera att det kommer en avslutande ')'
             readNextChar();          // läs förbi ')'
             return result;
      } else {
               // Fel! Högerparentes förväntades
      }
   } else if (nextIsNumber()) {
        return readNextNumber();
   } else {
       // Fel! Förväntade vänsterparentes eller tal
   }
}
```

Programmet är skrivet i ”pseudokod” dvs en del detaljer är utelämnade. Bl a bygger programmet på fyra primitiver: `nextToRead()` som returnerar nästa tecken *utan* att ta bort det från input-strömmen, `readNextChar()` som läser nästa tecken, `nextIsNumber()` som returnerar *true* om det som kommer går att tolka som ett tal annars *false* samt `readNextNumber()` som läser nästa tal. I Java kan de uttryckas t ex med standardklassen `StreamTokenizer` eller klassen `Stokenizer` som skrivits för att passa denna kurs. Se
`http://www.it.uu.se/edu/course/homepage/prog2/ht15/netLessons/tokenizer/`