DD2458, Problemlösning och programmering under press

Föreläsning 7: Syntaxanalys

Datum: 2007-10-30
Skribent(er): Erik Hammar, Jesper Särnesjö
Föreläsare: Mikael Goldmann

Denna föreläsning behandlade syntaxanalys. Syntaxanalys handlar om att utvinna hierarkisk data som är lagrad i ett linjärt format. Exempel på detta är programmerings- och markupspråk, naturliga språk, kemiska formler m.m.

Analysen kan delas in i två delar. Dels lexikal analys, som omvandlar indata till en sträng av tokens, och dels syntaxanalys, som bygger upp ett träd från strängen av tokens. För naturliga språk kan man säga att lexikal analys handlar om orden i en textsträng, och syntaxanalys om meningsbyggnaden.

1 Ett exempel

I detta exempel ska vi beräkna molekylvikten för en molekyl, givet dess kemiska formel och en lista med atomvikter. Molekylen vi använder är nitroglycerin, som har formeln $C_3H_5(NO_3)_3$. Atomvikterna för atomerna i molekylen hittar man i tabell 1.

Atom	Vikt
H	1
C	12
N	14
O	16

Tabell 1: Atomvikter

Molekylvikten är alltså $12 \cdot 3 + 1 \cdot 5 + (14 + 16 \cdot 3) \cdot 3 = 227$. Men hur skriver man ett program som räknar ut detta givet en godtycklig kemisk formel?

2 Lexikal analys

Den lexikala analysen för vårt exempel är tämligen enkel. $C_3H_5(NO_3)_3$ kommer här att skrivas som "C3H5(NO3)3$", där $ betyder slut på indata". Vi kommer alltså att använda de tokens som visas i tabell 2.

Alla tokens har en typ, och vissa tokens (*atom* och *num*) kommer även att ha ett värde. Med andra ord kommer indata att översättas till följande sträng av tokens: $atom(C)$, $num(3)$, $atom(H)$, $num(5)$, *lpar*, $atom(N)$, $atom(O)$, $num(3)$, *rpar*, $num(3)$, $.

Vi kan nu analysera dessa tokens med en kontextfri grammatik.

atom	En atom. Skrivs som en versal följt av noll eller en gemener.
num	Ett positivt heltal.
rpar	En högerparentes.
lpar	En vänsterparentes.
$	Slut på indata.

Tabell 2: Tokens

3 Kontextfria grammatiker

En kontextfri grammatik består av en mängd slutsymboler Σ (som exakt matchar den lexikala analysens tokens), en mängd ickeslutsymboler Γ, en startsymbol $< s > \in \Gamma$, samt en mängd av regler.

Slutsymbolerna ligger i löven för syntaxträden som grammatiken beskriver och går ej att expandera ytterligare. Ickeslutsymboler ligger i inre noder och expanderas till slutsymboler. Reglerna är på formen $VL ::= HL$, där VL består av exakt en ickeslutsymbol och HL består av ett godtyckligt antal slutsymboler eller ickeslutsymboler.

Idéen är att alla strängar som grammatiken beskriver går att skapa genom att utgå från $< s >$ och skriva om i ett eller flera steg med hjälp av reglerna.

Molekylerna i exemplet går att beskriva med följande grammatik:

$$
\begin{aligned}
< start > &::= < molekyl > \$ \\
< molekyl > &::= < multi >< molekyl > \\
< molekyl > &::= < multi > \\
< multi > &::= < singel > num \\
< multi > &::= < singel > \\
< singel > &::= lpar < molekyl > rpar \\
< singel > &::= atom
\end{aligned}
$$

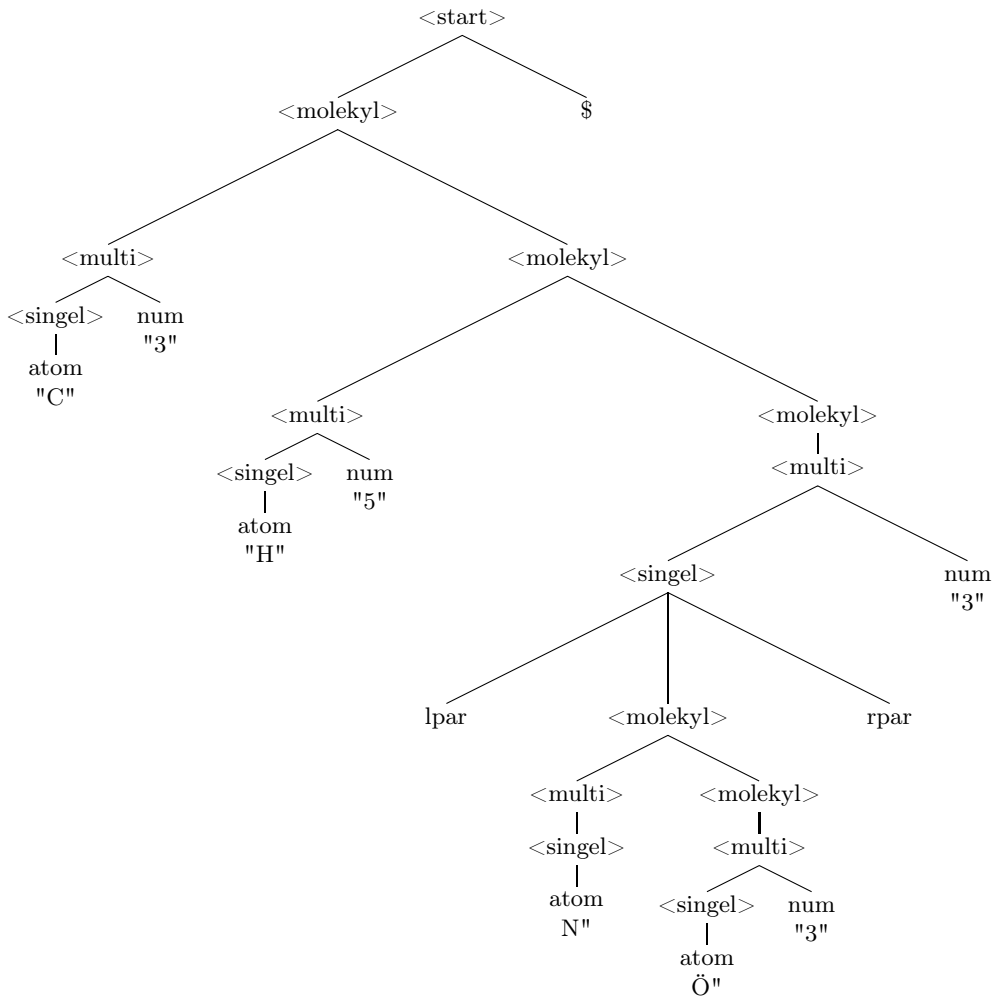
Om vi tillämpar denna grammatik på den sträng av tokens som den lexikala analysen gav oss får vi syntaxträdet i figur 1.

Vi kommer dock inte att bygga trädet explicit i detta exempel. För att beräkna molekylvikten ska vi använda rekursiv medåkning.

4 Syntaxanalys

4.1 Rekursiv medåkning

Rekursiv medåkning, eller *recursive descent*, är ett sätt att analysera en sträng av tokens i linjär tid. Rekursiv medåkning fungerar endast för de grammatiker där det är tillräckligt att man betraktar ett ändligt antal kommande tokens, s.k. $LL(k)$-grammatiker. Lyckligtvis är vår exempelgrammatik en $LL(1)$-grammatik, så vi behöver endast kolla på ett token i taget.



Figur 1: Syntaxträd

Tanken är att man skapar funktioner för ickeslutsymbolerna i grammatiken, som anropar varandra rekursivt. Alltså ska en funktion A matcha exakt de tokens som bygger upp en sträng som kan genereras av regeln $< a >$.

I vår implementation förutsätter vi att det finns två funktioner, peekToken, som returnerar aktuellt token i listan, samt nextToken, som flyttar fram läspekaren till nästa token. Vi förutsätter också att tokens har attributen *type* och *val*, samt att $vikt[x]$ är atomvikten för x.

```
START()
(1)   v ← MOLEKYL()
(2)   t ← peekToken()
(3)   if t.type = $
(4)       return v
(5)   else
(6)       ERROR!
```

```
MOLEKYL()
(1)   v ← MULTI()
(2)   t ← peekToken()
(3)   if t.type ∈ {$, rpar}
(4)       return v
(5)   else
(6)       return v + MOLEKYL()
```

```
MULTI()
(1)   v ← SINGEL()
(2)   t ← peekToken()
(3)   if t.type = num
(4)       v ← v · t.val
(5)       nextToken()
(6)   return v
```

```
SINGEL()
(1)   t ← PEEKTOKEN()
(2)   if t.type = atom
(3)      NEXTTOKEN()
(4)      return vikt[t.val]
(5)   else if t.type = lpar
(6)      NEXTTOKEN()
(7)      v ← MOLEKYL()
(8)      t ← PEEKTOKEN()
(9)      if t.type = rpar
(10)        NEXTTOKEN()
(11)        return v
(12)     else
(13)        ERROR!
(14)  else
(15)     ERROR!
```

4.2 Cocke-Younger-Kasamis algoritm

Alla grammatiker går ej att verifiera med rekursiv medåkning. Vi ska därför titta på Cocke-Younger-Kasamis algoritm, som fungerar för alla grammatiker som är skrivna på Chomsky-normalform. Detta innebär att alla regler är skrivna på någon av följande former, där $< a >$, $< b >$ och $< c >$ är ickeslutsymboler (som inte behöver vara distinkta) och x är en slutsymbol:

$$\begin{aligned} < a > &::= x \\ < a > &::= < b >< c > \end{aligned}$$

Alla kontextfria grammatiker går att beskriva på Chomsky-normalform. Exempelvis kan reglerna för $< singel >$ skrivas om på följande vis:

$$\begin{aligned} < singel > &::= atom \\ < singel > &::= < a >< b > \\ < a > &::= lpar \\ < b > &::= < molekyl >< c > \\ < c > &::= rpar \end{aligned}$$

Frågan vi vill besvara är huruvida en viss grammatik kan generera en given sträng $x_1..x_n$. Om grammatiken är skriven på Chomsky-normalform kan problemet delas upp i delproblem som avgör om en regel $< a_i >$ kan generera delsträngen $x_j..x_{j+k-1}$.

Det visar sig att dynamisk programmering är väl lämpat för att lösa detta problem. Vi använder en kub, gen, med booleska värden, där $gen[i, j, k]$ är sann om och endast om $< a_i >$ kan generera $x_j..x_{j+k-1}$. Svaret vi söker kommer att finnas i $gen[1, 1, n]$.

En snabb implementation av Cocke-Younger-Kasamis algoritm använder rekursion och memoisering (algoritmen nedan är en botten-upp-implementation). Algoritmens tidskomplexitet är dock $\Theta(n^3)$ i värsta fall, där n är antalet tecken i strängen som ska kontrolleras (under antagandet att grammatiken har konstant storlek), till skillnad från rekursiv medåkning, som går i linjär tid. Den bör därför endast användas om grammatiken kräver det.

```
COCKE-YOUNGER-KASAMI(x_1..x_n)
(1)   gen[i,j,k] ← false ∀i,j,k
(2)   for i = 1 to r
(3)     for j = 1 to n
(4)       if < a_i >::= x_j is a rule
(5)         gen[i,j,1] ← true
(6)   for k = 2 to n
(7)     for j = 1 to n − k + 1
(8)       for l = 1 to k − 1
(9)         foreach rule < a_i >::=< a_s >< a_t >
(10)          if gen[s,j,l] and gen[t,j + l,k − l]
(11)            gen[i,j,k] ← true
```

5 Reguljära uttryck

Reguljära uttryck är strängar som beskriver mängder av strängar. De har sin grund i formell språkteori, men används även i många verktyg och programmeringsspråk.

Reguljära uttryck definieras över något ändligt alfabet Σ, såsom bokstäverna A-Z, ASCII eller Unicode, och går att definiera med tre operationer på en mängd konstanter.

Konstanterna är samtliga tecken i Σ samt tecknet ϵ, som representerar en tom sträng. Operationerna är *alternering*, som matchar något av två mönster, *konkatenering*, som matchar ett mönster följt av ett annat, samt *Kleene-slutning*, som matchar noll eller fler förekomster av ett mönster. Detta summeras i tabell 3, där R är ett reguljärt uttryck och L_R mängden av strängar som R beskriver. Notera att R^i för Kleene-slutningar betyder i repetitioner av mönstret R.

De flesta implementationer av reguljära uttryck definierar ytterligare operationer. Dessa operationer tenderar att vara syntaktiskt sockersom gör det mycket bekvämare att skriva reguljära uttryck, men som inte låter uttrycken beskriva fler grammatiker, och som går att definiera med de tre redan nämnda operationerna. I tabell 4 ges ett par exempel på vanliga sådana operationer.

Betydelse	R	L_R
tom sträng	ϵ	$\{\epsilon\}$
något tecken $x \in \Sigma$	x	$\{x\}$
alternering	$(R_1)\|(R_2)$	$L_{R_1} \cup L_{R_2}$
konkatenering	$(R_1)(R_2)$	$\{ab \mid a \in L_{R_1}, b \in L_{R_2}\}$
Kleene-slutning	$(R)*$	$L_\epsilon \cup L_R \cup L_{R^2} \cup L_{R^3}$ ...

Tabell 3: Reguljära uttryck

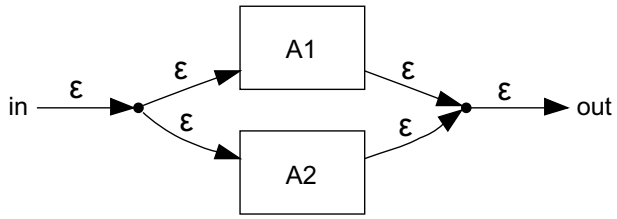
Betydelse	Uttryck	Ekvivalent uttryck
a, b, eller c	[abc]	a|b|c
något tecken mellan a och f, inklusivt	[a-f]	a|b|c|d|e|f
minst en förekomst av a	a+	aa*
högst en förekomst av a	a?	ϵ|a

Tabell 4: Exempel på utökad syntax för reguljära uttryck

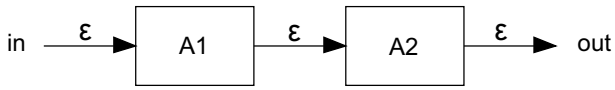
5.1 Reguljära uttryck som ändliga automater

Reguljära uttryck går även att beskriva som ändliga automater, där tillståndsövergångar representerar matchningar av ϵ alternativt $x \in \Sigma$.

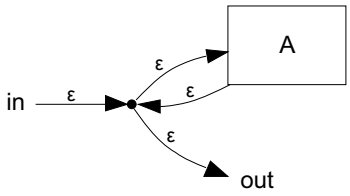
Man kan sedan bygga upp en automat som matchar ett reguljärt uttryck genom att kombinera automater som motsvarar de tre operationerna. Dessa tre automater går att definiera som i figur 2 - 4, där A, $A1$ och $A2$ är godtyckliga automater.



Figur 2: Alternering



Figur 3: Konkatenering



Figur 4: Kleene-slutning

5.2 Reguljära uttryck i Java

Javas standardbibliotek innehåller en implementation av reguljära uttryck, i paketet `java.util.regex`. Nedan följer ett exempel på hur man kan använda dem:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.Scanner;

public class pat2 {

  public static void main(String[] as) {
    // Beskriver en atom
    String atom = "[A-Z][a-z]?";

    // Beskriver höger- och vänsterparentes
    String paren = "[()]";

    // Beskriver positivt heltal
    String number = "[1-9][0-9]*";
```

```
    String s = as[0];

    // Skapa det reguljära uttrycket ( atom | paren | number )
    Matcher m =
      Pattern.compile(atom + "|" + paren + "|" + number).matcher(s);

    // Hitta nästa delsträng som matchar det reguljära uttrycket
    int pos = 0;
    while(m.find(pos)) {
      // Skriv ut delsträngen
      System.out.println(m.group());
      pos = m.end();
    }
  }

}
```

5.3 Reguljära uttryck i C/C++

Reguljära uttryck finns ej i standardbiblioteken för C eller C++. De kommer dock sannolikt att finnas med i nästa standard av C++, som har arbetsnamnet C++0x.