2D1458, Problemlösning och programmering under press

Föreläsning 7: Syntaxanalys

Datum: 2006-10-30
Skribent(er): John Lindberg & Oscar Sundbom
Föreläsare: Mikael Goldmann

1 Syntaxanalys

Den här föreläsningen tar upp tekniker som kan behövas för att utföra en syntaxanalys: lexikalanalys, grammatiker, rekursiv medåkning och reguljära uttryck.

1.1 Exempel på tillämpning

Beräkna molekylvikten för en molekyl givet dess kemiska formel (och kända atomvikter). Exempelvis har H_2O molekylvikten 18 eftersom H har atomvikt 1 och O har atomvikt 16.

En annan, lite mer komplicerad molekyl som vi kommer att använda vidare i senare exempel är nitroglycerin: $C_3H_5(NO_3)_3$ vilken har molekylvikt 227 (C har atomvikt 12 och N har atomvikt 14).

1.2 Lexikalanalys

För att kunna göra en syntaxanalys måste man först göra en lexikalanalys som omvandlar indatan till en ström av *tokens*. I det här fallet är indatan en textsträng, som för nitroglycerin har formen ”C3H5(NO3)3$”. $ symboliserar här ”slut på indata” och kan exempelvis vara en radbrytning.

Genom att analysera indatan kan man komma fram till att de tokens som förekommer i det här fallet är:

atom	- en atom. (Läses in som en versal bokstav följd av noll eller fler gemener.)
num	- ett tal.
lpar	- en vänsterparentes.
rpar	- en högerparentes.
$	- som ovan.

Motsvarande för ett vanligt språk skulle vara att ha ord och skiljetecken som tokens.

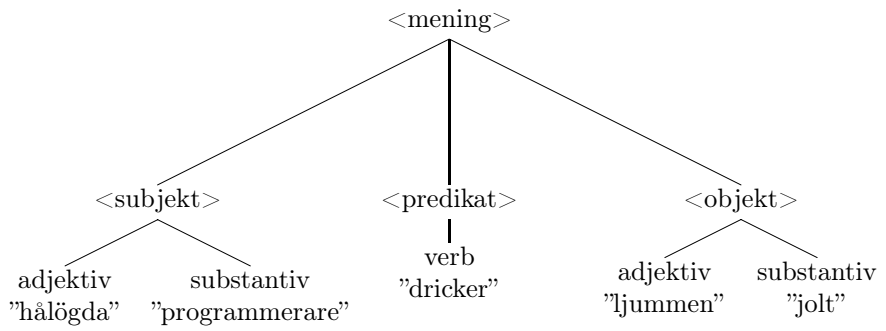
Att få in en tokenström som: atom num atom num lpar atom ... är i det här och många andra fall dock inte tillräckligt. Därför kan man knyta data till tokens som innehåller mer information och istället få något som:

atom	num	atom	num	lpar	atom	...
”C”	”3”	”H”	”5”		”N”	...

För att utföra lexikalanalysen kan man lämpligen använda reguljära uttryck, exempelkod för det kommer senare i anteckningarna.

1.3 Grammatiker

En grammatik används för att bygga ett syntaxträd. För en vanlig mening skulle till exempel ett syntaxträd kunna se ut så här:



Orden inom <. . . > kallas för ickeslutsymboler och orden inom ”. . . ” kallas för lexem. Orden som är direkt knutna till ett lexem kallas för slutsymboler. Indatan som trädet byggts av kallas fras.

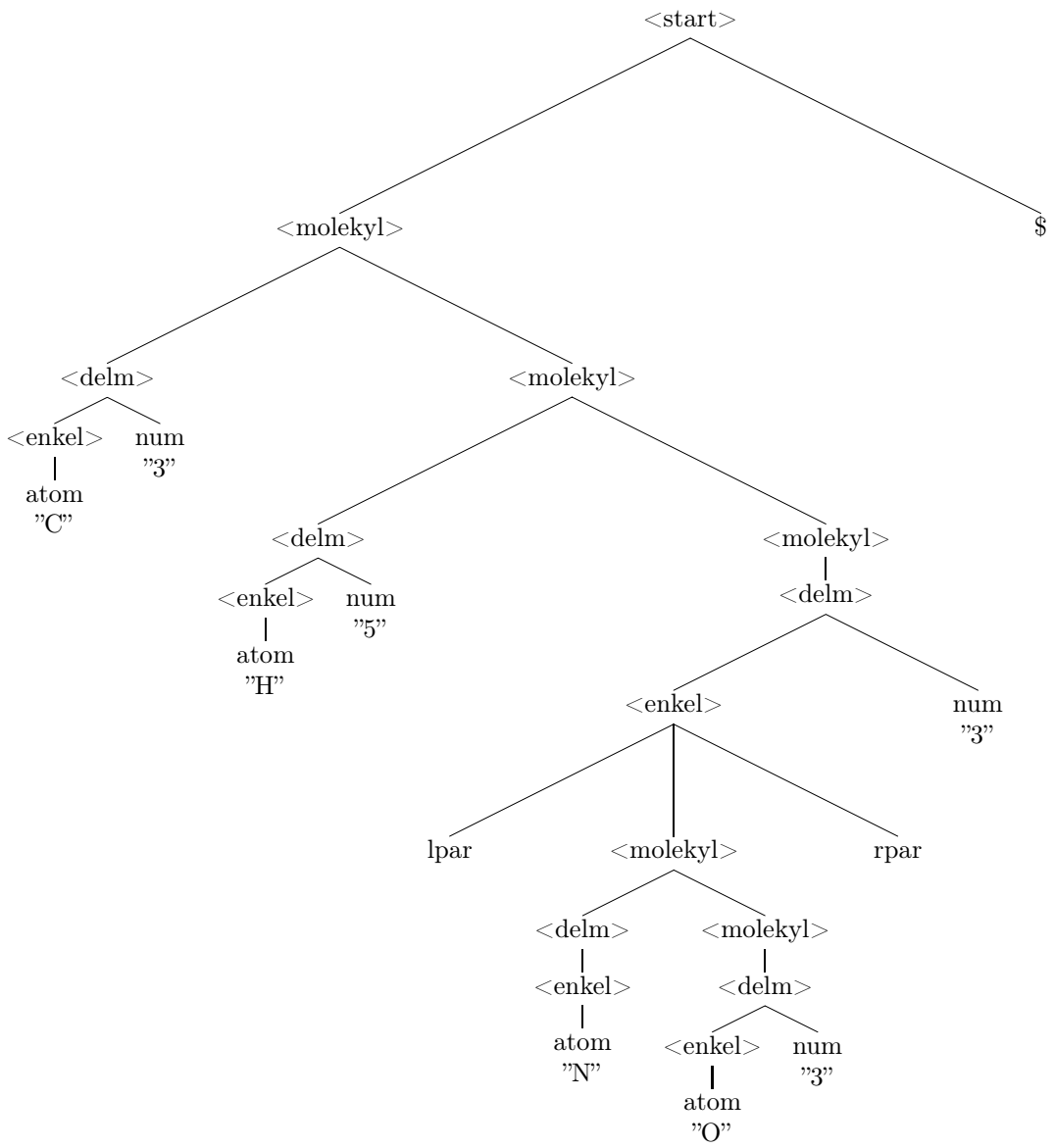
En grammatik har fyra beståndsdelar och kan uttryckas $<\Sigma, N, <start>, R>$. Σ är mängden av slutsymboler, vilket är samma sak som kallades tokens i lexikalanalysen. N är mängden av ickeslutsymboler, $<start>$ är startsymbolen $\in N$ (<mening> i exemplet ovan) och R är en uppsättning produktioner (eller regler) för hur ickeslutsymbolerna ser ut.

Produktionerna har formen $<a> ::= \beta \in (\Sigma \cup N)^*$ (där A^* betyder sträng av noll eller flera element ur A).

För molekylerna blir R så här:

```
   <start>  ::=  <molekyl>$
<molekyl>   ::=  <delm><molekyl>
<molekyl>   ::=  <delm>
   <delm>   ::=  <enkel>num
   <delm>   ::=  <enkel>
   <enkel>  ::=  atom
   <enkel>  ::=  lpar<molekyl>rpar
```

Syntaxträdet för nitroglycerin ser ut så här:



2 Rekursiv medåkning

Rekursiv medåkning (recursive descent) är ett sätt att bygga upp syntaxträd från en grammatik. Det fungerar inte för all grammatiker, men i de fall där det inte fungerar blir det ofta så komplicerat att man rekomenderas att använda ett färdigt verktyg för att generera koden som behövs.

Idén är att man skriver en metod för varje ickeslutsymbol som rekursivt anropar varandra. Den datastruktur som innehåller de tokens som man fick från lexikalanalysen behöver bara funktionalitet för att se ett aktuellt token (`peekToken()`) och för att byta till nästa token (`nextToken()`). För ett token innehåller `type` tokentypen och `val` den data som vi knutit till vissa tokens.

Förutsättningen för att det här ska fungera är att vi, givet vilket token vi står på och vilket som är nästa token, bara kan följa en produktion.

Observera att följande algoritm inte bygger trädet explicit utan beräknar molekylvikten direkt.

```
START()
(1)    w ← MOLEKYL()
(2)    t ← PEEKTOKEN()
(3)    if t.type ≠ $
(4)        ERROR()
(5)    else
(6)        output w
```

```
MOLEKYL()
(1)    w ← DELM()
(2)    t ← PEEKTOKEN()
(3)    if t.type ∈ {atom, lpar}
(4)        w ←W+ MOLEKYL()
(5)    else if t.type ∉ {rpar, $}
(6)        ERROR()
(7)    return w
```

```
DELM()
(1)    w ← ENKEL()
(2)    t ← PEEKTOKEN()
(3)    if t.type = num
(4)        w ← w · t.val
(5)        NEXTTOKEN()
(6)    return w
```

```
ENKEL()
(1)   t ← PEEKTOKEN()
(2)   if t.type = atom
(3)      w ← LOOKUPWEIGHT(t.val)
(4)      NEXTTOKEN()
(5)   else if t.type = lpar
(6)      NEXTTOKEN()
(7)      w ← MOLEKYL()
(8)      t ← PEEKTOKEN()
(9)      if t.type ≠ rpar
(10)        ERROR()
(11)     else
(12)        NEXTTOKEN()
(13)  else
(14)     ERROR()
(15)  return w
```

3 Allmäna grammatiker

Anta att vi har vår grammatik G på Chomsky normalfom (CNF). Detta innebär att den endast består av regler på nedanstående former:

```
<a>  ::=  x
<a>  ::=  <b><c>
```

Grammatiker med regler på annan form går att omvandla till CNF. Exempelvis:

```
 <a>  ::=  <b>x
        ↓
 <a>  ::=  <b><ny>
<ny>  ::=  x

 <a>  ::=  <b><c><d><e>
        ↓
 <a>  ::=  <b><ny1>
<ny1>  ::=  <c><ny2>
<ny2>  ::=  <d><e>
```

Givet indata $X_1...X_n$, kan G generera $X_1...X_n$ från startsymbolen $<a_0>$? Genom att använda regler på formen $<a_0>::=<a_1><a_2>$, kan vi dela upp problemet i delproblem: $X_1...X_i$ | $X_{i+1}...X_n$. Där vi vill att $<a_1>$ ska matcha första delen och $<a_2>$ andra.

Detta lämpar sig väl att lösas mha dynamisk programmering. För detta behöver vi en tredimensionell boolesk array: gen[j, i, k] $\leftrightarrow$ $<a_j>$ kan generera $X_i...X_{i+k-1}$. Matrisen blir ofta gles. En idé kan vara att istället göra en rekursiv implementation och memoisera.

```
DPCNF()
(1)   gen[j,i,k] ← false ∀ i,j,k
(2)   if X_i = b and ∃ regel <a_j>::=b
(3)      gen[j,i,1] ← true
(4)   for l = 2 to n
(5)      for i = 0 to n − 1
(6)         for r = 1 to l − 1
(7)            if ∃ regel <a_j>::=<a_s><a_t> så att
               gen[s,i,r] and gen[t,i + r,l − r]
(8)               gen[j,i,l] ← true
(9)   return gen[0,1,n]
```

4 Reguljära uttryck

Reguljära uttryck (över alfabetet Σ) skapas av följande:

R		$L_R \subseteq \Sigma^*$
a	$a \in \Sigma$	$L_a = \{a\}$
ε	”tomma strängen”	$L_\varepsilon = \{\varepsilon\}$
$(R_1)\|(R_2)$	union	$L_{(R_1)} \cup L_{(R_2)}$
$(R_1)(R_2)$	konkatenering	$L_{(R_1)(R_2)} = \{\alpha\beta \mid \alpha \in L_{(R_1)}, \beta \in L_{(R_2)}\}$
$(R)^*$	Kleene-slutning	$L_{(R)^*} = L_\varepsilon \cup L_R \cup L_{(R)(R)} \cup ... \cup L_{(R)^k}$

Av operatorerna ovan har Kleene-slutningen högst precedens, följt av konkatenering och slutligen union. Detta leder till att man ofta kan utelämna många parenteser. Många implementationer av reguljära uttryck tillåter också formler enl. tabellen nedan. Dessa gör inte att man kan representera fler grammatiker med reguljära uttryck, men det gör uttrycken mycket lättare att skriva.

Formel(exempel)	Innebörd
[abc]	Något av a, b eller c
[a-z]	Något av tecknen i intervallet (ofta efter teckenkod)
[^abc]	Något tecken som varken är a, b eller c

4.1 Reguljära uttryck i Java

Nedan följer ett exempel på hur man kan använda reguljära uttryck i Java.

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

public class pat2 {
    public static void main(String[] as)
    {
        String atom   = "[A-Z][a-z]?";
        String paren  = "[()]";
        String number = "[1-9][0-9]*";
        String s = as[0];
        Matcher m =
            Pattern.compile(atom + "|" + paren + "|" + number).matcher(s);
        int pos=0;
        while (m.find(pos)) {
            System.out.println(m.group());
            pos=m.end();
        }
    }
}
```