DD2458, Problemlösning och programmering under press

Föreläsning 7: Syntaxanalys

Datum: 2009-10-27
Skribent(er): Carl-Fredrik Sundlöf, Henrik Sandström, Jonas Lindmark
Föreläsare: Fredrik Niemelä

1 Syntaxanalys

Syntaxanalys går ut på att analysera en text och bestämma dess grammatiska struktur med avseende på en formell grammatik. Vi kan beskriva syntaxanalysen som en följd av olika steg. Följande bild beskriver de olika stegen.

Textsträng−(Lexikal analys)→ Tokenström−(Syntaxanalys)→ Syntaxträd −(Semantisk analys)→Resultat

Lexikal analys. Indata är en textsträng eller en textström. I det här steget delas indata upp i intressanta tokens. För t.ex. naturligt språk skulle den här delen gå ut på att dela upp en text i enskilda ord. Resultatet av det här steget representeras av en så kallad tokenström.

Syntaxanalys. Målet med det här steget är att identifiera de tokens vi skapade under den lexikala analysen och skapa ett syntaxträd. I exemplet med naturligt språk går det här steget ut på att grammatiskt analysera tokenströmmen och skapa grammatiska träd för meningarna i texten.

Semantiskanalys. Det här steget går ut på att bearbeta resultatet av syntaxanalysen till en mer lättläslig form. Det nya resultatet är menat att modellera ”betydelsen” eller ”resultatet”. Steget är inte nödvändigt men är ofta användbart eftersom det underlättar tolkning av resultatet.

2 Grammatiker

I syntaxanalyssteget använder vi oss av någon typ av grammatik för att kunna analysera vår tokenström. En grammatik innehåller en mängd slutsymboler och en mängd ickeslutsymboler. En slutsymbol är en sträng i indatan som inte går att bryta ned i delar utan att det förlorar sin betydelse. En ickeslutsymbol kan ses som typ i grammatiken, t.ex. substantiv för naturligt språk.

I den här kursen tas kontextfria grammatiker upp. Kontextfri betyder att en regel i grammatiken inte beror av kontexten i vilken regeln appliceras. I praktiken betyder detta att vänster ledet i produktionsreglerna endast innehåller en ickeslutsymbol. Så formen för produktionsreglerna blir:

<NT> = S

Där <NT> är en ickeslutsymbol och S är en eller flera ickeslutsymboler eller slutsymboler. Det här tolkas som att om vi har S i indatan så kommer vi skapa en nod i syntaxträdet med alla delar i S som barn. På det här sättet bygger vi upp syntaxträdet med hjälp av grammatiken.

3 Rekursiv medåkning

Istället för att rakt av konstruera ett syntaxträd kan vi använda oss av rekursiv medåkning. Algoritmen fungerar endast på klassen av LL(k) kontextfria grammatiker. $LL(k)$ står för Left to right, Leftmost derivation och k'et innebär att vi behöver titta k tokens i förväg för att kunna avgöra vilken produktionsregel vi ska använda.

En rekursiv medåknings implementation går i princip ut på att man konstruerar en funktion för varje ickeslutsymbol som det finns en produktionsregel för. Ofta är det så att det existerar fler än en enskild produktionsregel för en viss ickeslutsymbol. Man behöver då i funktionen titta k steg i förväg för att avgöra vilken av produktionsreglerna som skall utvecklas. Avsnitt 4.3 ger ett exempel på hur rekursiv medåkning fungerar.

4 Exempel - Molekylvikt

I följande exempel skall vi beräkna vikten för en molekyl givet molekylens kemiska formel samt en lista med atomvikter. Följande tabell visar atomernas vikt.

Atom	Vikt
H	1
O	16
N	14
C	12

Molekylvikten för t.ex. H_2O är $1 * 2 + 16 = 18$. Vi skall nu skapa ett program som beräknar dessa vikter. Problemet är att inte alla molekyler är så snälla som H_2O. Nitroglycerin har den kemiska formeln $C_3H_5(NO_3)_3$ och får då molekylvikten $12 * 3 + 1 * 5 + (14 + 16 * 3) * 3 = 227$. Så hur analyserar vi detta?

4.1 Lexikal analys

Indata till programmet för $C_3H_5(NO_3)_3$ är på formen C3H5(NO3)3. Den lexikala analysen kommer nu göra om indata till en ström av tokens. Vi kommer att ha följande tokens:

atom - En atom. En versal följt av noll eller en gemen blir en atom.
int - Ett tal.
lpar - En vänsterparentes.
rpar - En högerparentes.
$ - Slut på indata.
Vissa tokens kommer vi knyta ett värde till. I detta fall kommer *atom* och *int* få ett värde knutet till sig. Indata, $C_3H_5(NO_3)_3$, kommer att översättas till följande ström av tokens:

atom	int	atom	int	lpar	atom	atom	int	rpar	int	$
C	3	H	5		N	O	3		3	

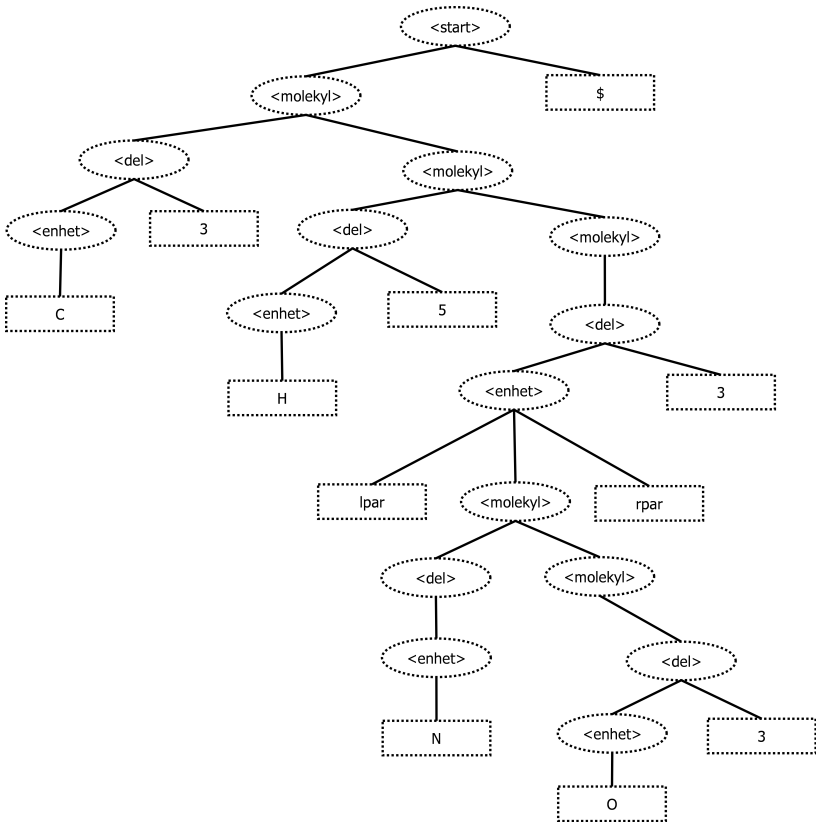
Strömmen av tokens kan nu analyseras med hjälp av en kontextfri grammatik.

4.2 Kontextfri grammatik

För att beskriva hur molekyler är uppbyggda kan vi använda oss av följande kontextfria grammatik:

<start>	=	<molekyl>$
<molekyl>	=	<del> <molekyl>
<molekyl>	=	<del>
<del>	=	<enhet> int
<del>	=	<enhet>
<enhet>	=	lpar <molekyl> rpar
<enhet>	=	atom

Genom grammatiken kan vi bygga följande syntaxträd för $C_3H_5(NO_3)_3$.



För att få reda på värdet av molekylvikten är det nu bara att traversera trädet och beräkna vikterna allteftersom. Att först bygga trädet och sedan traversera det är tidskrävande och det är i detta fall bättre att använda sig av rekursiv medåkning för att beräkna molekylvikten.

4.3 Rekursiv medåkning

Vi skapar nu funktioner för alla ickeslutsymboler i grammatiken. Vår grammatik är av typen *LL(1)* och vi behöver således titta på en token i förväg. Vidare förutsätter vi att det finns tre funktioner, peekToken() som kollar vad nästa token är, consume() som förbrukar en token samt slåUppAtomVikt() som retunerar atomvikten för en atom. Genom att funktionerna rekursivt kallar på varandra kommer vi beräkna molekylvikten allteftersom. En token har två attribut, *type* och *val*. Följande funktioner behövs.

```
START()
(1)     r = molekyl()
(2)     t = peekToken()
(3)     om t.typ = $
(4)         return r
(5)     annars
(6)         FEL!
```

```
MOLEKYL()
(1)     r = DEL()
(2)     t = peekToken()
(3)     om t.typ = atom, lpar
(4)         return r + molekyl()
(5)     annars om t.typ = $, rpar
(6)         return r
(7)     annars
(8)         FEL!
```

```
DEL()
(1)     r = ENHET()
(2)     t = peekToken()
(3)     om t.typ = int
(4)         consume()
(5)         return r * t.val
(6)     annars om t.typ = atom, lpar
(7)         return r
(8)     annars
(9)         FEL!
```

```
ENHET()
(1)    t = peekToken()
(2)    om t.typ = atom
(3)        consume()
(4)        return slåUppAtomVikt(t.val)
(5)    annars om t.typ = lpar
(6)        consume()
(7)        r = MOLEKYL()
(8)        t = peekToken()
(9)        om t.typ = rpar
(10)           consume()
(11)           return r
(12)       annars
(13)           FEL!
(14)   annars
(15)       FEL!
```

5 Cocke-Younger-Kasamis algoritm

Cocke, Younger och Kasamis konstruerade en algoritm som givet en kontextfri grammatik på Chomsky Normal Form (CNF), avgöra om en given sträng går att generera med den grammatiken.

För att en grammatik ska vara i CNF så måste dess produktionsregler vara på en av följande former:
$<l> = <r_1><r_2>$ där $<r_1>$ och $<r_2>$ är ickeslutsymboler.
$<l> = x$ där x är en slutsymbol.

Alla kontextfria grammatiker kan uttryckas i CNF. Om en regel är på formen
$<l> = x<r_1>$ där x är en slutsymbol och $<r_1>$ är en ickeslutsymbol skapar vi en ny ickeslutsymbol $<r_2>$ och inför produktionsregeln $<r_2> = x$ och substituerar så vi får:
$<l> = <r_2><r_1>$
$<r_2> = x$

På liknande sätt kan vi dela upp långa produktionsregler i mindre genom att införa nya ickeslutsymboler och substituera.

När vi har grammatiken i CNF kan vi använda dynamisk programmering för att lösa problemet genom att se på delproblemet om en delsträng av en viss längd kan genereras med reglerna eller inte.

Vi skapar en tredimensionell bool matris $gen[i, j, k]$ där i är index för regeln, j är startindex för delsträngen och k är längden på delsträngen. $gen[i, j, k]$ ska vara sann omm delsträngen som startar i index j och är k lång kan genereras med regeln

i, annars falsk.

Till att börja med sätts alla värden i gen till falska. Sedan går vi igenom och fyller i basfallen. Nämligen alla delsträngar av längden ett. För alla produktionsregler på formen $<l> = x$ där x är en slutsymbol letar vi efter tecken i indatan som är lika med x och sätter dessa poster i gen till sanna.

Sedan för alla möjliga längder av delsträngar börjar vi fylla i om det går att generera en delsträng av den längden genom att dela upp delsträngen med en produktionsregel på formen $<l> = <r_1><r_2>$.

Alltså, för varje längd av delsträng k, för varje startindex j och för varje uppdelning l av k som går att göra ser vi om vi kan hitta en regel $<l> = <r_1><r_2>$. Där vi kan generera $<r_1>$ ur intervallet j till $j+l$ och vi kan generera $<r_2>$ ur intervallet $j+l$ till $j+k$, om så är fallet sätter vi $gen[< l >, j, k]$ till sant.

Följande pseudokod beskriver algoritmen:

Algoritm 1: Cocke-Younger-Kasamis algoritm
COCKE-YOUNGER-KASAMIS$(x_1, x_2, \ldots, x_n)$

```
(1)   ∀i, j, k
(2)       gen[i, j, k] = false
(3)   ∀ regler < a_i >= S
(4)       ∀ tecken i strängen, (x_j)
(5)           om < a_i >= x_j
(6)               gen[i, j, k] = true
(7)
(8)     for k = 2 → n
(9)         for j = 1 → n − k + 1
(10)            for l = 1 → k − 1
(11)                ∀ < a_i >=< a_s >< a_t >
(12)                    if gen[s, j, l] och gen[t, j + l, k − l]
(13)                            gen[i, j, k] = true
```

Komplexiteten för algoritmen är ganska uppenbart $O(n^3)$ eftersom vi har tre nästlade for slingor som vardera kan köras $O(n)$ gånger. Algoritmen är avsevärt mycket långsammare än rekursiv medåkning men fungerar på alla kontextfria grammatiker.

6 Reguljära uttryck

Reguljära uttryck är en notation för att beskriva en mängd av strängar. Strängarna är uppbyggda av en mängd bokstäver från ett alfabet Σ. Ett specifikt reguljärt uttryck är en sträng som följer särskilda syntaxregler. Dessa regler varierar mellan olika implementationer men alla bygger i grunden på fyra olika byggstenar.
Dessa byggstenar är:

Konkatenering AB
Beskriver en särskild följd av uttryck. Efter A så måste B komma.
Alternativ A|B
Säger att antingen förekommer A eller så förekommer B.
Upprepning A*
Betyder att A får före komma ett godtyckligt antal gånger. A behöver inte förekomma alls.
Epsilon ϵ
Kommer inte konsumera någonting ur indata utan går vidare.

Med dessa byggstenar så kan man bygga alla andra reguljära uttryck. I java finns det som exempel en +-operator. Sätter man denna efter en bokstav, t.ex. a+, så betyder det att bokstaven måste förekomma minst en gång men samtidigt får den förekomma hur många gånger som helst. Denna operator kan lätt beskrivas med våra byggstenar som: aa*.

Några andra byggstenar som brukar finnas i de mest använda implementationerna är:
Gruppering (A)
Säger att A måste förekomma. Till exempel kan man ha problem med 'ö' och vilja matcha olika typer av 'ö' i ett ord. I ett ord som ström kan man göra det genom att byta ut 'ö' med str(ö|o|oe)m.
Hakparanteser [A]
Säger att någon av bokstäverna i A måste förekomma. Till exempel så kanske man vill matcha a, b eller c. Då skriver man [abc]. Hakparanteser brukar också innehålla stöd för att uttrycka spann av bokstäver. Om man vill ha alla bokstäver mellan a och z så kan man skriva [a-z].
Icke [^A]
Säger att inte någon av bokstäverna i A får förekomma.
Plus A+
Betyder att A måste förekomma minst en gång men får förekomma ett godtyckligt antal gånger.
Frågetecken A?
Betyder att A kan förekomma en gång men inte fler.

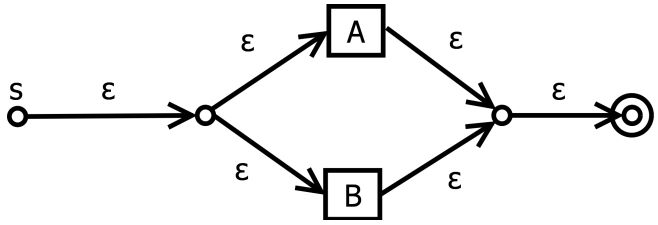
Förutom dessa så brukar det också finnas fördefinerade grupper av bokstäver. I t.ex. Java betyder \w samma sak som [a-zA-Z_0-9] och \s betyder
[\t\n\x0B\f\r] (ett mellanrum av någon typ).

6.1 Reguljära uttryck som ändliga automater

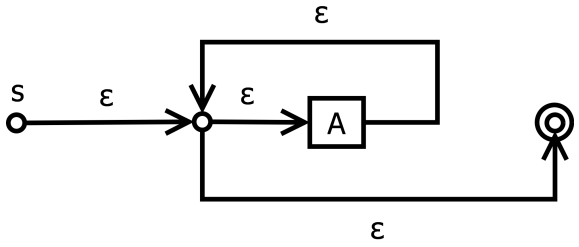
Om man inte har tillgång till ett färdigt bibliotek för reguljära uttryck men ändå vill evaluera en sträng med dessa så görs det bäst genom att ställa upp uttrycken som en ändlig automat. Sedan utvärderar man strängen genom att traversera automaten.

De tre första byggstenarna som beskrevs tidigare ser ut på följande sätt som automater:

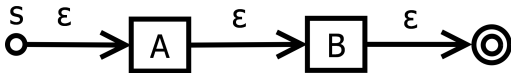
A|B



A*



AB



6.2 Reguljära uttryck i Java

För att få tillgång till java regex så använder man sig av klasserna Pattern och Matcher. Båda dessa ligger i paketet java.util.regex. Om man vill få allmän förklaring till vilka beteckningar man kan använda så hittar man en fin översikt av dessa på http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

Nedan följer ett exempel på hur man använder Pattern och Matcher. I exemplet så kompilerar man ett regex som matchar strängar som innehåller ett eller flera 'a'.

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

public class main {
    public static void main(String[] args) throws Exception {
        Pattern p = Pattern.compile("a+");
        Matcher m;
        m = p.matcher("aaaaaaaaa");
        if (m.matches())System.out.println("Good");
        else System.out.println("Bad!");

        m = p.matcher("aaaabbbb");
        if (m.matches())System.out.println("Good");
        else System.out.println("Bad!");
    }
}
```

Kör man programmet kommer outputen bli:

```
Good
Bad!
```

Lite mer avancerad regex i java:
"**[a-c]+[^def]***"
Matchar alla strängar som börjar med någon av bokstäverna a, b eller c en eller flera gånger och sedan följs av godtycklig sträng som inte innehåller d, e eller f.
Strängar som skulle matchas är: "abchej", "aaag", "a" och "apa".
Strängar som inte skulle matchas är: "hej", "abce", "alfa".

"**(hej|tja|hejsan)\\s+[A-Z][a-z]***"
Matchar strängar som börjar med "hej", "tja" eller "hejsan", följt av ett eller flera separeringstecken och sedan en sträng som börjar på en stor bokstav följt av ingen eller flera små bokstäver.
Strängar som matchas är: "hej Allan", "tja Urban".
Strängar som inte matchas är: "hej allan", "halloj Urban".

Om man har ett regex som inte fungerar så finns det olika verktyg för att felsöka det. Ett sådant är Regular Expression Test Page som går att finna på:
http://www.regexplanet.com/simple/index.html

6.3 Reguljära uttryck i C/C++

För att få tillgång till regex i C/C++ så kan man includera <regex.h>. regex.h finns som standard i GCC.

Det är inte så stor skillnad på regex i Java och i C/C++. Det som finns i Javas standardimplementation finns med största sannolikehet också i regex.h. Exemplerna i Java fungerar även om man portar dem till C/C++.

Här är ett kort exempel på hur regex kan användas i C++:

```
#include "regex.h"
#include <string>
#include <iostream>
using namespace std;

int main(){
    //Sätt upp ett pattern
    string pattern = "[h-k]ej";
    //Skapa och kompilera regex
    regex_t re;
    regcomp(&re, pattern.c_str(), REG_EXTENDED|REG_NOSUB);

    //Se om text kan matchas
    string text = "hej";
    if (regexec(&re, text.c_str(), (size_t)0, NULL, 0) == 0)
        printf("Hello World!\n");
```

```
    else
        printf("Bad!\n");

    return 0;
}
```

Tack till Johannes Svensson för hjälpen med exemplet.