Mästarprov Syntax

Emilie Ruixin Cao, ercao@kth.se

24 mars 2025

1 Grammatik

Grammatiken G definieras:

$$\begin{array}{llll} S & \to q & P & \to S \\ S & \to \textbf{ay } P & P & \to P * P \\ S & \to \textbf{ez } P & P & \to \textbf{?}P \\ S & \to S * S & P & \to \textbf{xt } P \\ S & \to \textbf{?}S & P & \to P\textbf{uv}P \end{array}$$

a) Visa att grammatiken G är tvetydig genom att göra två härledningar for en sträng u i L som resulterar i olika syntaxträd. Du behöver inte visa syntaxträden.

Ett exempel på en sträng som har två olika härledningar som också leder till två olika syntaxträd är strängen ”?q*?q*?q”.

Första härledningen (högerassociativt):

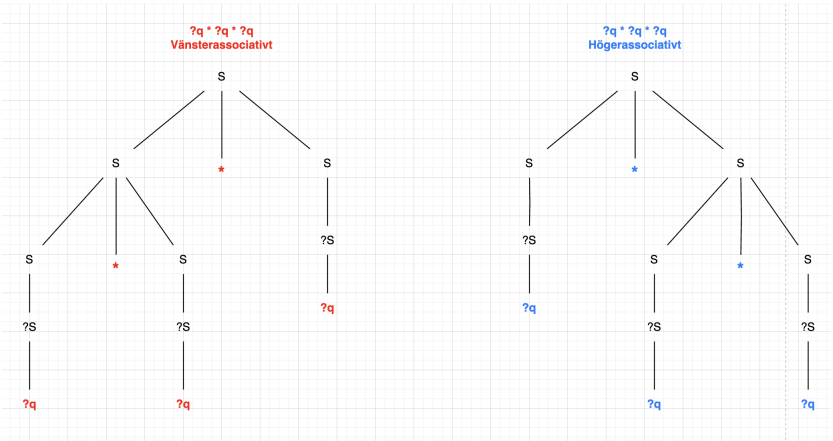
$$S \to S * S \to S * S * S \to S * S*?S \to S * S*?q$$
$$\to S*?S*?q \to S*?q*?q \to ?S*?q*?q \to ?q*?q*?q$$

Andra härledningen (vänsterassociativt):

$$S \to S * S \to S * S * S \to ?S * S * S \to ?q * S * S$$
$$\to ?q*?S * S \to ?q*?q * S \to ?q*?q*?S \to ?q*?q*?q$$

Dessa två härledningar resulterar i olika syntaxträd eftersom uttrycket S*S kan antingen byta ut det vänstra S:et mot ännu ett S*S så att det blir (S*S)*S eller så byts det högra S:et mot ännu ett S*S så att S*(S*S). Detta kommer att leda till två olika syntaxträd där den ena är en spegelbild av den andra. Anledningen till denna skillnad i syntaxträd beror på att regeln S*S kan tillämpas på två sätt. Antingen så expanderar man med det vänstra S:et först vilket leder till att uttrycket grupperas som (S*S)*S eller så expanderar man det högra S:et först så att uttrycket grupperas som S*(S*S). När det vänstra S:et expanderas först

blir syntaxträdet vänsterassociativt, medan expansion av högra S:et först gör trädet högerassociativt. Därmed blir det två olika syntaxträd och grammatiken är tvetydig enligt boken sida 128. De två olika syntaxträden visas i bilden nedan.



b) Du ska nu göra en grammatik för ett språk som är en utökning av L med slutsymboler för vänsterparentes, "(", och högerparentes, ")".

Startsymbolen i denna grammatik är '<s>'.

$$
\begin{aligned}
&< s > ::= \ '(' < q >< s' > \ ')' \\
&< s' > ::= \ ' ' \mid \ '*' < q >< s' > \\
&< q > ::= \ CAPS \mid \ 'ay \ ' < p > \mid \ 'ez \ ' < p > \mid \ '?' \ < s > \\
&< p > ::= \ '(' < t >< p' > \ ')' \\
&< p' > ::= \ ' ' \mid \ '*' < t >< p' > \mid \ 'uv \ ' < t >< p' > \\
&< t > ::= < i > \mid \ '(' < s > \ ')' \\
&< i > ::= \ '?' < p > \mid \ 'xt' < p >
\end{aligned}
$$

CAPS: [A-Z]+ i den definierade grammatiken. Alltså en eller flera kapitaliserade bokstäver.

Ovanstående grammatik är icke-tvetydig och kan parsas med en rekursiv medåkare med endast en lookahead eftersom de olika utfallen har en unik terminal som 'förutspår' vilken icke-terminal som ska köras härnäst. Dessutom så är alla produktioner ej vänsterrekursiva då vänsterrekursion kan leda till oändliga loopar vid rekursion. Ex. kan $S \rightarrow S * S$ leda till oändliga loopar då den första produktionen S pekar på är sig själv och därmed finns risken att den aldrig terminerar. Enligt drakboken s.68 så kan man eliminera vänsterrekursion genom följande teknik: Givet regeln $A \rightarrow A\alpha$ så kan det skrivas om till två

regler $A \rightarrow \beta R$ och $R \rightarrow \alpha R \mid \epsilon$.

c) Motivera kort varför en hypotetisk implementation av en parser med rekursiv medåkning utgående från din grammatik i b) endast behöver inspektera en slutsymbol (token) framåt i en sträng för att avgöra vilken regel som ska användas för en given ickeslutsymbol.

Alla utfall i min grammatik i b) kan parsas genom att inspektera en slutsymbol fram eftersom alla olika utfall har en unik slutsymbol som första token. Exempelvis om man börjar på $< s >$ och parsern sedan läser av $< q >$ först så kommer den kunna välja mellan fyra alternativ varav varje alternativ börjar på ett unikt ASCII-tecken. Alltså går det att avgöra nästa operation bara baserat på en slutsymbol framåt då ex. slutsymbolen *?* i produktionen för $< q >$ innebär att nästa operation är $< s >$ eller om nästa slutsymbol är *a* och då är nästa produktion $< p >$. Sedan kommer den också evaluera $< s' >$. $< s' >$ kan också välja mellan två olika alternativ med unika slutsymboler: den tomma strängen eller stjärnan (*). Läser den av stjärna så kommer den evaluera $< q >$ igen och sedan $< s' >$ med rekursiva anrop tills den når den tomma strängen och terminerar. Produktionen i $< p >$ fungerar på samma sätt förutom att $< p' >$ kan evaluera tre olika fall: antingen börjar den en produktion med **uv**, eller en produktion med stjärnan *****, eller så är det tomma strängen. Om $< p' >$ inte leder till tomma strängen så kommer det göras ett anrop på $< t >$ som antingen läser av en parentes och startar produktionen $< s >$ inom parentesen, annars kallar den på $< i >$ som då läser av om det är **xt** eller **?** som nästa symbol. Då kallar den på $< p >$ igen. I övrigt så är kallas $< p' >$ rekursivt i regeln $< p' >$ EFTER anropet på $< t >$.

För att detta ska funka och inte vara tvetydigt så följer det att den inte kan vara vänsterrekursiv eftersom detta innebär att den kan fastna i en oändlig rekursion. Tekniken som används för att eliminera vänsterrekursion härleds på sådant sätt att om man har A $\rightarrow$ Aα | β som är vänsterrekursiv så kan man skriva om den på formen A $\rightarrow$ βR, R $\rightarrow$ αR | ϵ. Denna teknik förklaras i drakboken på sida 68. Enligt drakboken (s.223) så behöver tre krav uppfyllas för att det ska vara LL(1) – alltså en rekursiv medåkare med en lookahead.

Första kravet är att för ett uttryck A $\rightarrow$ α | β ska varken α eller β kunna härleda en sträng som börjar på samma slutsymbol *a*. Detta stämmer i den konstruerade grammatiken i b) eftersom alla möjliga utfall leder till strängar som har en unik slutsymbol först. Exempelvis så leder $< s' >$ ::= $''$ | $'*' < q >< s' >$ till att den ena produktionen ger ϵ som första slutsymbol och den andra produktionen ger att '*' är nästa slutsymbol. Detsamma syns i produktionen för $< p' >$ där ϵ, *****, och **uv** är de tre olika första slutsymbolerna.

Det andra kravet för uttrycket är att max en av produktionerna α eller β får producera en tom sträng vilket också stämmer då ex. $< s' >$ kan härleda en tom sträng endast från ena ledet som den består av. Detsamma gäller för $< p' >$ också där det bara är en av produktionerna som ger tomma strängen. För $< t >$ så är det ingen produktion som ger tomma strängen vilket också uppfyller villkoret då det är MAX en produktion som kan leda till tomma strängen. Alltså

uppfylls även krav två i enlighet med drakbokens tre krav på en LL1 grammatik.

Den tredje regeln säger att om en grammatikregel har mer än en produktion och en av produktionerna ger den tomma strängen så måste de resterande produktionernas första slutsymbol vara disjunkt från huvudets följande symbol. Alltså för uttrycket A $\rightarrow \alpha \mid \beta$ där $\beta \rightarrow \epsilon$ så måste FIRST(α) vara disjunkt från FOLLOW(A). Om vi kollar på exempelvis $< s' >$ där den första produktionen ger ϵ så är FIRST($'* < q >< s' >$) = {*}. FOLLOW($< s' >$) blir symbolen som kommer efter produktionen $< s' >$. Produktionen $< s' >$ förekommer endast i regeln $< s > ::= '(' < q >< s' > ')'$. Detta innebär att FOLLOW($< s' >$) alltid kommer att vara en högerparantes ')' då $< s' >$ är sluten i stängda paranteser. Då $< s >$ också är startsymbolen lär FOLLOW($< s' >$) också innehålla '$' som markerar slutet på input. Vi kan se att FOLLOW($< s' >$) inte innehåller '*' och är skilt från FIRST($'* < q >< s' >$). Alltså är de disjunkta och kommer vara kompatibla med en LL(1) parser. Samma argument kan dras för $< p' >$ (där epsilon kan härledas i en av produktionerna) som har FIRST($'* < t >< p' >$) = '*' och FIRST('uv' $< t >< p' >$) = 'u'. FOLLOW($< p' >$) förekommer endast i regeln $< p > ::= '(' < t >< p' > ')'$ och där blir FOLLOW($< p' >$) = ')' då denna också är sluten inom parentes. Vi kan se att dessa två mängder är disjunkta. Alltså är grammatiken tvetydig och kan parsas med en rekursiv medåkare som har en lookahead. Enligt drakboken så kan endast grammatiker som är **entydiga** parsas med en LL(1) parser och därmed följer även att grammatiken är entydig.

d) Ge ett exempel på en sträng v i ditt språk från b), och en sträng w från L, sådana att om alla vänsterparenteser och högerparenteser tas bort från v så erhålls w.

Ett exempel på en sträng v i mitt språk från b) är strängen '(ay((q)))', om man tar bort alla vänster- och högerparanteser från v så erhålls strängen w 'ayq' från språket L. Härledningar för båda språken ges nedan:

I språket L blir strängen w = 'ayq':

$$S \rightarrow ayP \rightarrow ayS \rightarrow ayq$$

Vi kan alltså se att strängen w = 'ayq' går att härleda för språket L. Detta motsvarar strängen v = '(ay((q)))' utan paranteser i min grammatik från b). För att härleda att strängen v accepteras i min grammatik görs följande härledning:

$$\begin{aligned}
&< s > \rightarrow (< q >< s' >) \rightarrow (ay < p >< s' >) \rightarrow (ay(< t >< p' >) < s' >) \\
&\rightarrow (ay((< s >) < p' >) < s' >) \rightarrow (ay((< q >< s' >) < p' >) < s' >) \\
&\rightarrow (ay((q < s' >) < p' >) < s' >) \rightarrow (ay((q\epsilon) < p' >) < s' >) \\
&\rightarrow (ay((q)\epsilon) < s' >) \rightarrow (ay((q))\epsilon) \rightarrow (ay((q)))
\end{aligned}$$

Här ser vi att man kan härleda strängen v = '(ay((q)))' med hjälp av grammatiken från uppgift b). Tar vi bort alla vänster- och högerparanteser så återstår endast strängen 'ayq' vilket motsvarar strängen w från språket L.

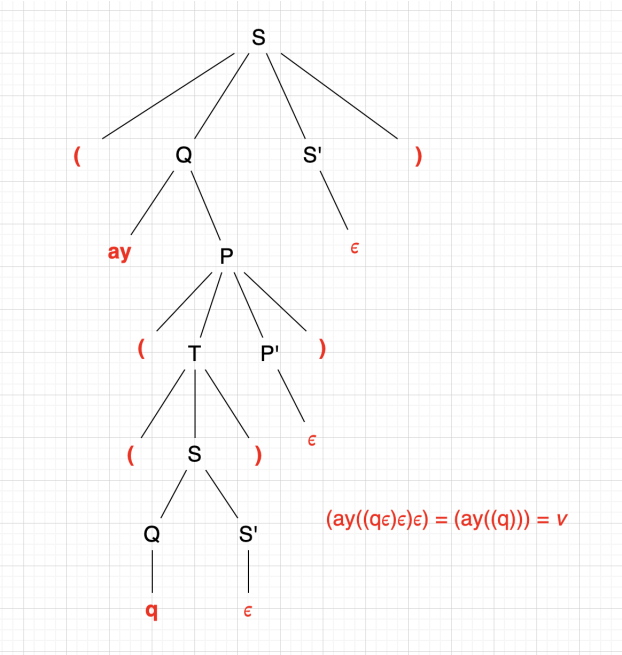
e) Ge en härledning i din grammatik från b) som visar att strängen v från d) verkligen tillhör språket som din grammatik definierar. För att undvika långa upprepningar i härledningen kan du använda tydligt definierade förkortningar.

En härledning av strängen v visas i motiveringen för uppgift d), men nedan lyder denna härledning igen:

$$
\begin{aligned}
&< s > \rightarrow (< q >< s' >) \rightarrow (ay < p >< s' >) \rightarrow (ay(< t >< p' >) < s' >) \\
&\rightarrow (ay((< s >) < p' >) < s' >) \rightarrow (ay((< q >< s' >) < p' >) < s' >) \\
&\rightarrow (ay((q < s' >) < p' >) < s' >) \rightarrow (ay((q\epsilon) < p' >) < s' >) \\
&\rightarrow (ay((q)\epsilon) < s' >) \rightarrow (ay((q))\epsilon) \rightarrow (ay((q)))
\end{aligned}
$$

f) Ge ett syntaxträd för strängen v i d) och motivera kort varför det bara finns ett enda syntaxträd för v i din grammatik. Det räcker att du använder namnen på dina tokens i trädet, dvs du behöver inte ange deras representation som sekvenser i ASCII.

Ett syntaxträd för strängen v = '(ay((q)))' ges av följande:

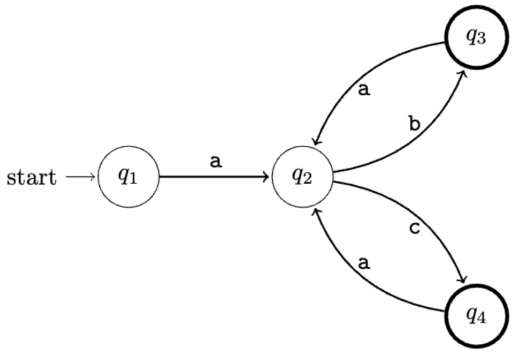


Strängen (ay((q))) har endast ett unikt syntaxträd eftersom grammatiken är entydig och LL(1)-kompatibel. Varje steg i härledningen är entydigt bestämt av den aktuella lookahead-terminalen, tack vare att grammatiken har disjunkta FIRST-mängder och korrekt hanterar ϵ-produktioner med FOLLOW-mängder. Till exempel, när parsern ser (, ay eller q, finns det alltid bara ett möjligt val för nästa produktion. Detta säkerställer att strängen endast kan härledas på ett sätt, och därmed finns det bara ett syntaxträd.

2 Automater och Reguljära Uttryck

Vi betraktar språk över alfabetet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ och definierar följande operationer och konstanter som returnerar sådana språk:

$$
\begin{aligned}
R(x, L_1) &= \{w \mid xw \in L_1\} \text{ för } x \in \Sigma \\
S(L_1) &= L_1^* \\
K(L_1, L_2) &= \{uv \mid u \in L_1 \text{ och } v \in L_2\} \\
P(L_1, L_2) &= \{u \mid u \in L_1 \text{ eller } u \in L_2\} \\
Z(L_1, L_2) &= \{uv \mid \texttt{cb}u \in L_1 \text{ och } v \in L_2\} \\
B &= \{\texttt{ab}\} \\
C &= \{\texttt{ac}\}
\end{aligned}
$$



Låt M vara språket som känns igen av DFA:n över Σ med följande grafiska representation.

a) Uttryck språket $Z(L, M)$ endast i termer av operationerna R, S, K, P, och språken B, C och L. Med andra ord, ge en definition av $Z(L, M)$ med hjälp av B, C och L som inte använder mängdnotation eller andra språkoperationer än R, S, K och P.

Språket M kan beskrivas på följande vis med hjälp av de definierade operatorerna R, S, K, och P samt språken B, C, och L:

$$M = K(P(B, C), S(P(B, C)))$$

Den grafiska representationen av DFA:n visar att språket M accepterar strängar som börjar på antingen 'ab' eller 'ac'. Utöver detta så kan strängarna fortsätta med noll eller flera iterationer av antingen 'ab' eller 'ac'. Regeln K tar in två språk som parametrar och returnerar ett språk som består av strängarna uv där u tillhör det första språket som input och v tillhör det andra språket som input. I detta fall är det P(B,C) som är det första språket som är input vilket innebär att den första delsträngen u ska tillhöra språket som alla strängar som både finns med i språket B och språket C. Detta innebär att u kommer att vara antingen 'ab' (språket B) eller 'ac' (språket C). Det andra språket som är input är S(P(B,C)) vilket innebär att delsträngen v kommer att bestå av noll eller flera iterationer av antingen 'ab' eller 'ac' då detta är språket som returneras från P(B,C). Kleenestjärnan i operatoren S innebär att alla strängar i inputspråket kan konkateneras noll eller flera gånger. Om man då sedan applicerar operatorn K på dessa på inputspråk så får man alltså en konkatenering mellan 'ab' eller 'ac' och ab* eller ac*.

Med detta i åtanke kan man alltså uttrycka Z(L,M) på följande vis:

$$Z(L, M) = K(R(b, R(c, L)), K(P(B, C), S(P(B, C))))$$

Det vi gör är ännu en konkatenering med K. Denna gång är det första inputspråket $R(b, R(c, L))$ och det andra inputspråket är M, som definierats

tidigare. Eftersom $K(L_1, L_2)$ definieras som mängden av alla konkateneringar där den första delsträngen u tillhör L_1 och den andra delsträngen v tillhör L_2, betyder detta att den andra delsträngen v i konkateneringen bör tillhöra språket M. Det första inputspråket i K, alltså $R(b, R(c, L))$, betecknar alla strängar i språket L som börjar på bokstäverna 'cb'. Detta inputspråk kan delas upp i en inre och yttre operator:

1. Inre operatorn: $R(c, L)$
$R(c, L)$ innehåller alla strängar u sådana att OM vi lägger till ett 'c' framför u, så tillhör cu språket L. Detta betyder att $R(c, L)$ innehåller exakt de strängar i L där vi kan anta att ett 'c' redan har lästs bort.

2. Yttre operator: $R(b, R(c, L))$
$R(b, R(c, L))$ innehåller alla strängar u sådana att OM vi lägger till ett 'b' framför u, så tillhör bu mängden $R(c, L)$. Eftersom vi redan vet att $R(c, L)$ består av alla u där $cu \in L$, så betyder detta att bu måste vara en sådan sträng att $cbu \in L$.

När man då använder operatorn K för konkatenering så kan man alltså applicera detta som det första inputspråket för att definiera att den första delsträngen måste tillhöra språket L samt börja med bokstäverna 'cb'. Det andra inputspråket är våran definition av M som vi kan se ovan.

b) Antag att språket L känns igen av DFA:n $(Q, \Sigma, \delta, q, F)$. Ge en matematisk definition av en DFA som känner igen språket $\{u \mid \texttt{cb}u \in L\}$. Med matematisk definition av en DFA menas en definition av en tupel $(Q', \Sigma, \delta', q', F')$ och inte en grafisk representation.

En matematisk definition av en DFA som känner igen språket är:

$$L' = \big(Q, \Sigma, \delta, s, F\big)$$
$$s = \delta(\delta(q, c), b)$$

Denna DFA känner igen språket $\{u \mid \texttt{cb}u \in L\}$ eftersom starttillståndet (den fjärde parametern i tupeln) ska börja i ett tillstånd där den redan har 'gått igenom' bokstäverna 'cb'. För att kunna säkerställa detta så kan man applicera det boken kallar för *transition function* δ (s.48) två gånger. Funktionen δ har två inparametrar: ett tillstånd och en symbol. Alltså är tillståndet δ(a, q) det tillståndet p man når när man övergår från tillstånd q till p med en kant markerad med 'a' emellan dem. Detta är relevant till denna matematiska definition av L' som känner igen språket $\{u \mid \texttt{cb}u \in L\}$ eftersom vi måste säkerställa att starttillståndet innefattar att strängen har 'cb' som startsymboler. Först applicerar vi δ på symbolen c och tillståndet q som är starttillståndet för språket L. Genom denna applikation av tillståndsfunktionen får vi ett nytt tillstånd q_1 i vilket vi har 'ackumulerat' ett 'c' i strängen redan. Från detta

tillstånd $q_1 = \delta(q, c)$ så applicerar vi δ igen fast med bokstaven 'b' och på det nya tillståndet q_1. Alltså $\delta(q_1, b)$ vilket ger oss det nästa tillståndet vi når om vi ackumulerar ett 'b' i vår sträng. Då $\delta(\delta(q, c), b)$ returnerar det tillstånd vi är på då vi redan ackumulerat 'cb' i starten av strängen så motsvarar det alltså starttillståndet till L'. Sedan är det samma mängd tillstånd Q som i L:s definition, samma alfabet $\sum = \{a, b, c\}$ som från L, samma funktion δ som från L, och samma accepterande tillstånd F som från L. Detta då det nya språket är uppbyggt på samma sätt som språket L, det viktigaste att skilja på är starttillståndet. I detta fall är starttillståndet det tillstånd där vi redan har checkat att de första två symbolerna ska vara 'cb'. Vi kan kalla detta nya starttillstånd för $s = \delta(\delta(q, c), b)$.

c) Antag att din DFA från b) är sådan att alla tillstånd i F' har en övergång för a till ett loopande tillstånd $q_0 \in Q' \setminus F'$ (dvs q_0 är ett icke-accepterande tillstånd som har övergångar till sig själv för alla $x \in \Sigma$). Förklara kortfattat hur du med hjälp av denna DFA kan bygga en DFA som känner igen $Z(L, M)$.

Eftersom Z(L,M) består av en konkatenerad sträng med två delsträngar från olika språk är det lämpligt att koppla ihop DFA:n för L' och DFA:n för M. Detta kan göras med hjälp av något som kallas *product* automaton enligt boken s.43. Det som händer är att varje tillstånd representeras av en tupel (p,q) där p är ett tillstånd från L':s DFA och q är ett tillstånd från M:s DFA. Starttillståndet kan då exempelvis skrivas som en tupel $(q_{l_{start}}, q_{m_{start}})$ istället för bara $q_{l_{start}}$. För samtliga tillstånd q som tillhör DFA:n till L' så kan man beteckna tillstånden med tupeln $(q, q_{m_{start}})$. Alltså att medan vi behandlar strängen cb$u$ som tillhör språket L' så kommer samtidigt M:s DFA alltid konstant förbli i ett starttillstånd $q_{m_{start}}$. När vi sedan når ett accepterande tillstånd i DFA:n för L' så kommer det finnas en övergång markerad 'a' till tillståndet q_0 i DFA:n för L', men samtidigt så når den även tillståndet q_2 i DFA:n för M (se figuren på M för tydliggörande). Då får vi alltså tupeln (q_0, q_2) när vi initialt når tillståndet q_0 i DFA för L'. För alla resterande tillstånd i DFA:n för M så kommer tillståndet i DFA:n för L' vara q_0. Så exempelvis när DFA:n för M vandrar från q_2 till q_3 så kommer tupeln gå från (q_0, q_2) till (q_0, q_3). På samma sätt kan man resonera för hela DFA:n för M. Samma övergångar i DFA:n för M som finns på figuren i uppgiftsbeskrivningen tillhandahålls i den nya DFA:n för Z(L,M) men varje tillstånd definieras som en tupel där q_0 är konstant för DFA:n för L' medan tillståndet i M:s DFA skiftar. På så sätt kan man bygga upp en DFA som känner igen Z(L,M), men då måste man som sagt använda sig av det så kallade *product automaton*. Den informationen tillhandahålls i boken s.43-44.

3 Språkklasser

Låt Σ_1 och Σ_2 vara alfabet, och låt f vara en funktion från Σ_1^* till Σ_2^* sådan att $f(uv) = f(u)f(v)$ för alla $u \in \Sigma_1^*$ och $v \in \Sigma_1^*$. Antag

att L är ett reguljärt språk över Σ_2. Antag vidare att L' är ett språk över Σ_1 som definieras av en (kontextfri) grammatik. Vilken är den minsta språkklass som tagits upp i kursen som språket $\{w \in \Sigma_1^* \mid f(w) \in L$ eller $w \in L'\}$ då med säkerhet tillhör? Motivera ditt svar noggrant.

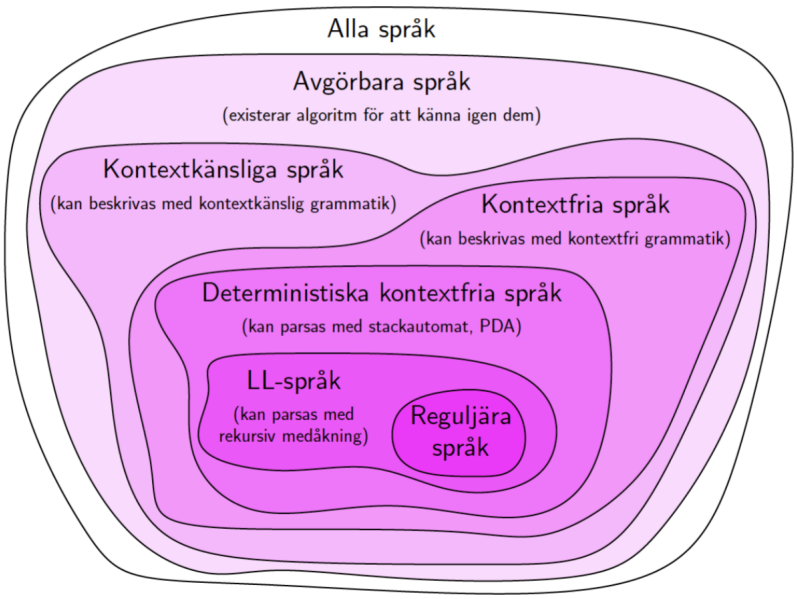
Språket S = $\{w \in \Sigma_1^* \mid f(w) \in L$ eller $w \in L'\}$ tillhör med säkerhet språkklassen 'kontextfria språk'. Detta beror på att funktionen f är en så kallad homomorfism. Definitionen ges i kursboken s.141.

Definitionen av en homomorfism ges av boken som: Funktionen h är en homomorfism på alfabetet $\sum$ där $w = a_1a_2a_3...a_n$ är en sträng med symboler från $\sum$ om $h(w) = h(a_1)h(a_2)...h(a_n)$. Vi kan därmed se att funktionen f är en homomorfism då vi kan se att för $w = uv$ så $f(w) = f(uv) = f(u)f(v)$. Enligt bokens definition så är reguljära språk slutna under en homomorfism ("/... the regular languages are closed under homomorphism", s.141). Mer formellt står det i boken: "If L is a regular language over alphabet $\sum$, and h is a homomorphism on $\sum$, then h(L) is also regular."

Detta innebär att i vårt fall, där L är givet som ett reguljärt språk i uppgiften, så innebär det att f(L) också är ett reguljärt språk. Om man sedan kollar vidare i boken så står det att inversen av en homomorfism också bevarar ett reguljärt språk. Enligt boken definieras inversen till en homomorfism h på följande vis: ".../ $h^{-1}(L)$ /.../ is the set of strings w in $\sum^*$ such that *h(w)* is in L."I vårt språk S så säger vi att antingen så $f(w) \in L$ eller $w \in L'$. Uttrycket $f(w) \in L$ kan kopplas bokens definition av invers homomorfism så att $f^{-1}(L)$ är mängden strängar w i $\sum_2^*$ så att $f(w)$ tillhör språket L. Alltså definierar det första uttrycket $f(w) \in L$ ett reguljärt språk i enlighet med boken.

Om man sedan evaluerar det andra påståendet i eller-uttrycket, $w \in L'$, så kan man genast konstatera att detta uttryck definierar ett kontextfritt språk eftersom det är givet i uppgiften att L' är definierat av en kontextfri grammatik. Enligt föreläsningen 'syntax 4' på slide 12 så står det att kontextfria språk är en språkklass som består av alla språk som kan beskrivas med en kontextfri grammatik vilket precis är vad L' är. Därmed kan man konstatera att alla strängar så att $w \in L'$ betecknar ett språk av klassen kontextfritt språk.

Eftersom språket S definieras av ett 'eller'-uttryck så är språket S en union mellan $f(w) \in L$ och $w \in L'$, då kan man se det som en union mellan ett reguljärt språk och ett kontextfritt språk. Enligt Wenn-diagrammet från föreläsningen 'syntax 4' slide 12 så är reguljära språk en delmängd av kontextfria språk. Detta innebär alltså att unionen mellan språken med säkerhet kommer att vara ett *kontextfritt* språk.


Alla språk
Avgörbara språk
(existerar algoritm för att känna igen dem)
Kontextkänsliga språk
(kan beskrivas med kontextkänslig grammatik)
Kontextfria språk
(kan beskrivas med kontextfri grammatik)
Deterministiska kontextfria språk
(kan parsas med stackautomat, PDA)
LL-språk
(kan parsas med rekursiv medåkning)
Reguljära språk