Examensarbete D, 20p

AUTLAW - AUTomata and LAnguage Workshop

Slutrapport

Författare:
Daniel Holmgren, c96dhn@cs.umu.se

Handledare:
Frank Drewes

13 oktober 2003

Abstract
The AUtomata and LAnguage Workshop is a tool developed to help understanding the concepts of formal languages, the meaning of finite automata and their abilities to specify languages. AUTLAW also helps understanding the connections between deterministic finite automata and non-deterministic finite automata and gives the possibility to convert between them, minimize them and to examine a string to see if it is part of at specific language. The AUTLAW system lets the user easily create, edit, examine and compare automata.

Innehåll

1 Inlednig

Varje student som läser datavetenskap lär sig ändliga automater som ett av de mest grundläggande och vitala inom datalogi. De ändliga automaterna är ofta även den första kontakten med abstrakta maskiner och datamodeller. Trots att konceptet med formella språk är enkelt, brukar det dock ofta skapa problem för studenten; man är inte van vid det abstrakta tankesättet. För att underlätta detta, skulle det vara bra med en miljö i vilken studenten själv kan "leka" med automater; enkelt skapa sina automater, undersöka dem, konvertera dem och så vidare.

Då kan man ställa sig frågan: finns det några bra programmiljöer för att skapa, editera och experimentera med ändliga automater? Åtminstone ett system finns på internet: **JFlap**

JFlap

JFlap är ett system som berör inte mindre än sex olika delmoment inom formella språk; ändliga automater, pushdown-automater, Turing-maskiner med både ett och två band, grammatikkonvertering samt konvertering av reguljära uttryck. Problemet är dock att det kanske är lite *för* avancerat för en nybörjare. JFlap förutsätter lite mer kännedom om både grammatiker, språk och automater och lämpar sig kanske inte för de som ännu inte förstått begreppen.

Om man koncentrerar sig på ändliga automater-delen känns denna lite rörig. Användargränssnittet är inte speciellt intuitivt och funktionerna känns svåra att lära sig och förstå sig på, vilket förmodligen kan vara en aning frustrerande för någon som inte kan speciellt mycket om automater. Det råder dock inga som helst tvivel om att JFlap, för den som vill och orkar sätta sig in i programmet ordentligt, är ett utmärkt verktyg att fördjupa sina kunskaper inom formella språk.

Ett annat, enklare, system skulle därför vara idealiskt; ett system som inte kräver så mycket förkunskaper och som har ett användargränssnitt som är enkelt och behagligt att använda.

Varför ändliga automater?

Nu kan man ju fråga sig: varför är ett system med ändliga automater speciellt lämpligt för att lära sig formella språk? Svaret på det är att utifrån en pedagogisk synpunkt är det viktigt att systemt kan analysera det studenten-/användaren gör, exempelvis måste korrekthet kunna avgöras genom någon typ av test. Dessutom måste systemet kunna väcka lite nyfikenhet hos användaren genom att visa att intressanta saker kan göras, exempelvis förenkling eller minimering av en automat. Eftersom ändliga automater ofta är det första en student ser av formella språk (man får inte glömma att automater kan ses som riktade grafer; något som ofta behandlas i inledande matematikkurser) är dessa därför idealiska för ändamålet.

Något som är viktigare än det faktum att studenten kanske stött på riktade grafer tidigare är, som tidigare nämnts, att utbudet av intressanta saker att göra är tillräckligt stort. Detta förutsätter algoritmer som kan användas på automaterna, exempelvis algoritmer för att konverta en icke-deterministisk automat till en deterministisk dito eller att minimera en automat[1]. Eftersom ändliga automater inte kan ta hand om *alla* typer av språk är det betydligt enklare att finna lämpliga algoritmer till dessa än till exempelvis mer kraftfulla automater som exempelvis *pushdown-automater* (det vill säga kontextfria språk) där utbudet är starkt begränsat. Exempelvis är det oavgörbart om två PDA accepterar samma språk.

Vad behövs i systemet?

För att vara ett användbart system behövs inte minst ett intuitivt och användbart användargränssnitt. Med det skall användaren enkelt kunna skapa sina automater med tillstånd och transitioner för att sedan enkelt kunna manipulera värden och utöka funktionaliteten. För att vara ett pedagogiskt system behövs funktioner som på ett enkelt och pedagogiskt sätt hjälper användaren att förstå vad som händer. Exempelvis skall användare kunna skapa en automat som motsvarar ett reguljärt uttryck och därefter jämföra dessa. Systemet skall då, internt, konvertera och minimera automaterna och jämföra dem med varandra samt visa resultat. Är resultaten olika skall användaren ges exempel på strängar som finns i det ena men inte i det andra språket.

Användare skall även kunna följa spårningen av en sträng genom en automat vilket systemet bör visa genom att i grafen markera det/de tillstånd som spårningen för tillfället befinner sig i. Man skall efter avslutad spårning enkelt se huruvida den inmatade strängen accepteras eller förkastas av automaten och därmed inte återfinns i språket.

Grundfrågeställning

För att kunna skapa ett så lärorikt med ändå enkelt system som möjligt måste de existerande algoritmerna undersökas och utvärderas huruvida de överhuvudtaget går att implementera (inom en rimlig tidperiod). Detta är något som är alldeles utmärkt för en teoretisk fördjupning, vilket ju innebär att sätta sig in i den existerande litteraturen för att förstå sig på detaljerna och använda dem i något syfte. I det här fallet är syftet att skapa ett pedagogiskt verktyg som hjälper studenten att förstå konceptet med formella språk; åtminstone de reguljära språken. Därför måste de algoritmer som förekommer i en mängd litteratur, och som redan bevisats tillräckligt många gånger, undersökas i syfte att underlätta implementeringen av systemet.

[1] Dessa begrepp kommer att förklaras i detalj i senare kapitel

Innan skapandet av systemet kan påbörjas behövs således en frågeställning att ta hänsyn till:

- VAD skall implementeras? Vilka funktioner finns det? Hur används de?
- HUR skall dessa implementeras? Hur kan algoritmer skrivas om och anpassas?

För att kunna skapa ett system som är pedagogiskt och lärande behövs en fördjupning i de teoretiska algoritmer som existerar för ändliga automater. Eftersom ändliga automater är så pass enkla att använda finns en uppsjö av olika typer av algoritmer för att konvertera mellan i princip varje typ av ändlig automat som finns och för att jämföra två olika automater för att se huruvida de motsvarar samma språk. Inom ramen för arbetet och med tanke på systemets syfte visade sig fyra algoritmer vara av särskilt intresse.

- Algoritmer för att jämföra en automat med en annan.
- Algoritmer för att utifrån reguljära uttryck skapa en automat som accepterar samma språk.
- Algoritmer för att konvertera olika typer av automater.
- Algoritmer för att testa huruvida en viss sträng accepteras av en automat eller inte.

Den observante läsaren noterar att exempelvis algoritmer för att konvertera en reguljär grammatik till en automat och vice versa inte finns med. Anledningen till detta är att någonstans måste man sätta stopp och att ta med dessa algoritmer ansågs inte tidsmässigt intressant. Det är dock viktigt att komma ihåg att reguljära grammatiker och ändliga automater också hänger ihop på samma sätt som reguljära uttryck och automater. För den som är intresserad av att konvertera grammatiker till automater och vice versa kan prova exempelvis tidigare nämnda JFlap. Detta går att ladda ner från internet på [5].

I rapporten kommer varje algoritm att noggrannare synas och med hjälp av de algoritmer och definitioner som existerar kommer implementerbara funktioner att arbetas fram. Rapporten kommer även att ytligt beskriva hur det färdiga systemet är uppbyggt samt hur systemets GUI fungerar.

Först kommer dock läsaren att få en grundläggande kurs i vad som menas med språk och strängar, vad reguljära uttryck är samt hur automater fungerar och beskrivs; detta beroende på att dessa är mycket grundläggande och vitala delar och egentligen det som hela systemet kretsar kring. Utan denna vetskap kommer resten av rapporten att inte säga speciellt mycket. För den redan invigde kan det bli en nyttig och trevlig repetition av kunskaperna.

2 Introduktion till reguljära språk

Detta kapitel ger läsaren en kort men koncis introduktion till reguljära språk och begrepp som används inom detta område, exempelvis strängar och språk, reguljära uttryck samt ändliga automater.

2.1 Strängar och språk

Som systemets namn anger, AUTomata and LAnguage Workshop, vilket kan översättas till "automat- och språkverkstad", behandlar systemet just automater och språk. Men vad menas egentligen med "språk"? Det finns många olika typer av språk, till exempel naturliga språk, datorspråk och matematiska språk. Man kan dock ge alla språk en gemensam definition; ett språk är en mängd av strängar över ett alfabet.

Nu är det dock inte speciellt intressant med naturliga språk eftersom dessa är mycket komplexa. Det som istället är intressant är enkla strängar över enkla alfabeten på vilka rekursiva definitioner och mängdoperationer kan appliceras.

Definition 2.1 (Strängar och språk)

Låt Σ vara en ändlig mängd av symboler, kallad *alfabet*. Mängden Σ^* av alla *strängar över* Σ utgörs av alla ändliga sekvenser bestående av element i Σ. Ett *språk* är en delmängd av Σ^*.

Enligt definitionen är alltså en sträng över en mängd tecken en ändlig sekvens av dessa och är det mest fundamentala i ett språk. Den mängd ur vilken elementen i strängen väljs, *alfabetet*, är ändlig med alla tecken olika. Varje sträng i ett språk, naturligt eller ej, brukar betraktas som odelbart vilket till exempel innebär att symbolen *lokal* inte kan delas upp i *lo* och *kal* eller att symbolen *format* har varken med *for* eller *mat* att göra. Dessa är alla individuella "tecken" i alfabetet och en sträng över detta alfabet är en sekvens av ord - en mening. Alfabetet hos ett programmeringsspråk består av reserverade ord, variabler och symboler för språket och en sträng är helt enkelt en bit kod i språket.

För att definiera ett språk måste restriktioner för vilka strängar som skall ingå finnas. Ett språk är enligt definitionen en delmängd av alla tillgängliga strängar Σ^*. Denna restriktion kan göras på en mängd olika sätt och för de reguljära språken, som är de som är intressanta här, görs det vanligen med reguljära uttryck, grammatiker eller ändliga automater. Alla dessa tre är sätt är likvärdiga och centrala delar i detta examensarbete. I synnerhet reguljära uttryck och automater kommer att på ett något mer utförligt sätt definieras och förklaras nedan. Reguljära grammatiker lämnas till läsaren att på egen hand undersökas, till exempel i [4], [2] eller [3]. Anledningen till att dessa utelämnas kommer att klargöras i senare kapitel.

2.2 Reguljära uttryck

Ett reguljärt uttryck är ett sätt att beteckna ett visst språk. Detta görs med hjälp av tre enkla operationer: konkatenering, union samt *Kleene star*-operationen. Konkatenering är en sammanslagning av två deluttryck, en union är ett val mellan två deluttryck och Kleene star är en speciell operation som betecknar noll eller fler förekomster av ett deluttryck (eller enstaka input-symboler).

Eftersom union och konkatenering är associativa behövs inga parenteser i sekvenser med dessa operatorer. För att ytterligare minska antalet parenteser har en prioritetsordning införts hos operatorerna där *Kleene star* har högst prioritet och därefter konkatenering och sist union. Utöver dessa operatorer finns även andra som är vanligt förekommande i reguljära uttryck, som till exempel $\boldsymbol{u}^+$ som används som förkortning för $\boldsymbol{uu}^*$. Dessutom skrivs ibland $\boldsymbol{uu}$ som $\boldsymbol{u}^2$, $\boldsymbol{u}^2\boldsymbol{u}$ som $\boldsymbol{u}^3$ och så vidare.

Ett reguljärt uttryck är ett mönster som definierar ett visst språk och en specifik sträng tillhör språket om och endast om den matchar detta mönster. Operatorerna i ett uttryck ger olika möjligheter att konstruera språk. Konkatenering specificerar till exempel ordning; $\boldsymbol{ab}$ innebär att ett $\boldsymbol{a}$ alltid måste följas av ett $\boldsymbol{b}$, *Kleene star* specificerar repetition; $\boldsymbol{a}^*$ tillåter repetetion av $\boldsymbol{a}$ godtyckligt många gånger och union specificerar val; $\boldsymbol{a} \cup \boldsymbol{b}$ tillåter val mellan $\boldsymbol{a}$ eller $\boldsymbol{b}$.

Definition 2.2 (Operatorer i reguljära uttryck)

Låt L och L' vara två språk. Då gäller följande:

$$\begin{array}{ll} LL' = & \{uv \mid u \in L, v \in L'\} \\ L^* = & \{u_1 \dots u_k \mid k \in \mathbb{N}, u_1 \dots u_k \in L\} \\ L \cup L' = & \{u \mid u \in L \vee u \in L'\} \end{array}$$

Exempel
Betrakta följande två reguljära uttryck:

$$(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^* \text{ och } (\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*$$

Det första uttrycket representerar alla strängar innehållande följden $\boldsymbol{aa}$ och det andra alla strängar med följden $\boldsymbol{bb}$. Kombineras dessa uttryck med $\cup$-operatorn erhålls uttrycket

$$(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{aa}(\boldsymbol{a} \cup \boldsymbol{b})^* \cup (\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{bb}(\boldsymbol{a} \cup \boldsymbol{b})^*$$

som representerar alla strängar innehållande $\boldsymbol{aa}$ eller $\boldsymbol{bb}$. Här har alltså de två deluttrycken kombinerats ihop med union för att på så sätt bilda det slutliga uttrycken. Språket som uttrycket betecknar kommer att bestå av alla de strängar som matchar uttrycket.

2.3 Automater och språk

Överallt i vår vardag omgärdas vi av maskiner och automater. Gemensamt för alla automater är att en viss input bearbetas och ger något slags output. En varumaskin tar pengar som input och returnerar mat eller dryck eller motsvarande. Ett kombinationslås förväntar sig som input en viss kombination av siffror och öppnar låset om kombinationen är den rätta. De abstrakta maskiner som introduceras här tar en sträng som input och ger som output ett svar på huruvida strängen ingår i ett visst språk eller ej. Dessa maskiner kallas för *ändliga automater* (eng. *finite state machines, finite automata*), förkortat FA.

Ändliga automater

En ändlig automat är en abstrakt maskin som inte tar någon hänsyn till vilken typ av hårdvara som är involverad. Dess input är en sekvens av symboler (språkets alfabet) och dess output är ett positivt eller negativt svar på huruvida strängen är en del av språket eller ej. En FA kan således ses som en "språkaccepterare" (eng: *language acceptor*), till skillnad mot ett reguljärt uttryck som istället skapar strängar som språket består av. Det är dock viktigt att komma ihåg att reguljärt uttryck och ändliga automater är likvärdiga.

En ändlig automat brukar ofta betecknas med en märkt, riktad graf, kallad *tillståndsdiagram*. Dessa kommer att behandlas senare.

Man brukar vanligtvis skilja mellan två olika typer av ändliga automater; den *deterministiska* automaten (förkortad **DFA**) och *icke-deterministiska* automater (förkortat **NFA**). Som namnen anger är en **DFA** mycket riktigt deterministisk, dvs från varje tillstånd finns alltid exakt en väg för varje input-symbol. I en **NFA** kan det ibland finnas flera alternativ att välja för en och samma input-symbol. Skillnaderna mellan **DFA** och **NFA** kommer att klargöras i kommande kapitel.

Definition 2.3 (Deterministisk automat)

En deterministisk ändlig automat kan ses som en kvintuppel bestående av

- En ändlig mängd tillstånd, Q.
- Ett specifikt tillstånd, $S \in Q$, kallat *starttillstånd*
- En mängd tillstånd, $F \subseteq Q$, kallade *sluttillstånd* eller *accepterande tillstånd*
- En ändlig mängd input-symboler Σ, språkets alfabet
- samt till sist en *transitionsfunktion*, $\delta : Q \times \Sigma \rightarrow Q$, som anger hur automaten byter tillstånd.

Syftet med automaten är alltså att bestämma huruvida input-strängen accepteras och därmed finns med i språket som automaten representerar. Transitionsfunktionen är en funktion som tar två argument; aktuellt tillstånd samt en input-symbol och returnerar ett tillstånd som är nästa tillstånd för automaten. Rent tekniskt ger transitionsfunktionen svar på frågan: "Vilket tillstånd kommer jag till om jag står i Q och följer kanten märkt med en viss symbol?"

Det är viktigt att komma ihåg att för en **DFA** spelar tidigare beräkningar och tillståndsbyten ingen som helst roll. Det enda som är viktigt är vilket tillstånd automaten för närvarande befinner sig i och hur den återstående strängen ser ut. För varje tecken som läses kommer automaten att ändra sitt tillstånd och detta brukar betecknas med $[q_i, aw] \vdash_{\mathrm{M}} [q_j, w]$. Denna beteckning indikerar att konfigurationen $[q_j, w]$ (vilket innebär att automaten befinner sig i tillståndet $q_{j]}$ och har strängen w kvar att läsa) erhålls från konfigurationen $[q_i, aw]$, detta i och med att $\delta(q_i, a) = q_j$. Tecknet $\vdash_{\mathrm{M}}$ kan utläsas som *ger* eller *producerar* (eng: *yields*). $\vdash^*$ betecknar en beräkning som består av noll eller flera produktioner.

Exempel:
En **DFA** M definieras enligt nedan och accepterar alla strängar över $\{a, b\}$ som innehåller delsträngen bb. Detta språk kan beskrivas som ett reguljärt uttryck enligt $L(M) = (\boldsymbol{a} \cup \boldsymbol{b})^* \boldsymbol{bb} (\boldsymbol{a} \cup \boldsymbol{b})^*$.

$$\begin{array}{lll} \mathrm{M}: & Q = & \{q_0, q_1, q_2\} \\ & \Sigma = & \{a, b\} \\ & S = & q_0 \\ & F = & \{q_2\} \end{array}$$

Transitionsfunktionen δ är given och visas i tabellform, kallat *transitionstabell*. Tillstånden listas vertikalt och alfabetet horisontellt. Nästa tillstånd för ett givet tillstånd q_i och ett läst tecken a fås genom att i tabellen finna skärningen mellan raden för q_i och kolumnen för a.

δ	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

Beräkningarna för M med input-strängarna *abba* och *abab* ser ut enligt:

$[q_0, abba]$	$[q_0, abab]$
$\vdash [q_0, bba]$	$\vdash [q_0, bab]$
$\vdash [q_1, ba]$	$\vdash [q_1, ab]$
$\vdash [q_2, a]$	$\vdash [q_0, b]$
$\vdash [q_2, \varepsilon]$	$\vdash [q_1, \varepsilon]$
accepteras	förkastas

Automaten accepterar alltså strängen *abba* medan den förkastar *abab* som ju inte innehåller delsträngen *bb*. Detta kan utläsas ur tabellen eftersom q_2 är ett accepterande tillstånd medan q_1 inte är det.

Hur är då en **DFA** kopplad till ett speciellt språk? Det fastställs med följande definition:

Definition 2.4 (Språket för en DFA)

Låt $M = (Q, \Sigma, \delta, S, F)$ vara en **DFA**. *Språket* för M, betecknat L(M), består av alla strängar u sådana $[S, u] \vdash^* [q, \varepsilon]$ för något $q \in F$.

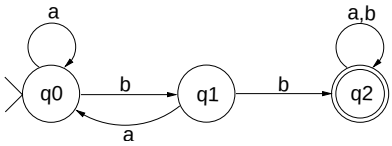
Tillståndsdiagram

Som tidigare nämnts är ett tillståndsdiagram en riktad, märkt graf där hörnen betecknar tillstånden hos automaten medan kanterna betecknar transitionsfunktionerna. Ett tillståndsdiagram är betydligt mer intuitivt och naturligt och kommer företrädesvis att beteckna en automat.

I diagrammet betecknar hörnen alltså tillstånden vilka ritas som en rund ring. De speciella tillstånden, start- och sluttillstånd, betecknas med en pil respektive dubbla ringar.

En transition i en **DFA** representeras av en kant i tillståndsdiagrammet och för att spåra kontrollen av en input-sträng skapar man en väg genom grafen som börjar i S. Eftersom automaten är deterministisk finns för varje inputsträng en unik väg genom grafen och som tidigare nämnt accepteras strängen om beräkningen slutar i ett accepterande tillstånd.

Exempel:
Tillståndsdiagrammet till automaten ovan ser ut enligt

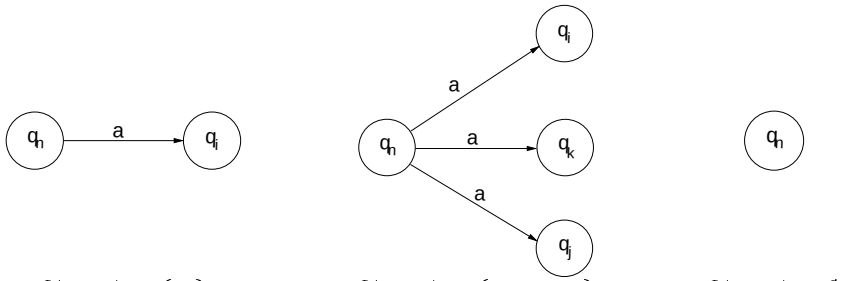


Hur ska man göra om man istället för en automat som accepterar alla strängar som innehåller substrängen *bb* istället vill ha en automat som accepterar alla strängar som *inte* innehåller *bb*? Det är faktiskt mycket enklare än man tror. Den automaten är nämligen precis densamma som den som accepterar *bb* men där accepterande och icke-accepterande tillstånd "byter plats", det vill säga alla accepterande tillstånd i den gamla maskien blir icke-accepterande den nya och tvärtom.

Icke-deterministisk automat

Den automat som tidigare studerats, **DFA**, har egenskapen att för varje tillstånd $q_i \in Q$ och varje tecken $a \in \Sigma$ finns alltid exakt en transitition från q_i märkt a, vilket gjort att man *alltid* direkt kan avgöra vilken väg i grafen man ska ta vid ett inläst tecken. Nu skall denna egenskap frångås och istället introduceras en *icke-deterministisk ändlig automat*, kallad **NFA**.

En transition hos en **NFA** har samma syfte som hos en **DFA**: att ändra automatens tillstånd baserat på aktuellt tillstånd och inläst tecken. I en **NFA** behöver man dock nödvändigtvis inte byta till ett specifikt tillstånd utan man har, för samma inlästa tecken, möjlighet att välja bland noll eller flera tillstånd. Nedan visas tre olika sätt att grafiskt representera transitioner i en **NFA**.



$\delta(q_n, a) = \{q_i\}$ $\delta(q_n, a) = \{q_i, q_j, q_k\}$ $\delta(q_n, a) = \emptyset$

Förutom transitionsfunktionen δ är en **NFA** identisk med en **DFA**.

Relationen mellan en **NFA** och en **DFA** kan sammanfattas i "varje deterministisk ändlig automat är en icke-deterministisk". Transitionsfunktionen hos en **DFA** innehåller exakt ett tillstånd för varje tillstånd och input-symbol. För en **NFA** tillåts ingen, en eller flera tillstånd för. Rent formellt kan det skrivas som:

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

där $\mathcal{P}(Q)$ betecknar mängden av alla delmängder av Q.

Exempel:
Givet en automat M nedan. En input-sträng *ababb* genererar (minst) tre olika beräkningar. Beräkningar i en **NFA** betecknas på samma sätt som för en **DFA** med tecknet $\vdash$.

M: $Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$F = \{q_2\}$
$S = q_0$

δ	a	b
q_0	$\{q_o\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

$[q_0, ababb]$
$\vdash [q_0, babb]$
$\vdash [q_0, abb]$
$\vdash [q_0, bb]$
$\vdash [q_0, b]$
$\vdash [q_0, \varepsilon]$

$[q_0, ababb]$
$\vdash [q_0, babb]$
$\vdash [q_1, abb]$

$[q_0, ababb]$
$\vdash [q_0, babb]$
$\vdash [q_0, abb]$
$\vdash [q_0, bb]$
$\vdash [q_1, b]$
$\vdash [q_2, \varepsilon]$

Den andra beräkningen i automaten M stannar efter exekvering av tre instruktioner eftersom det inte finns någon information om vad som ska hända när automaten befinner sig i tillstånd q_1 och läser ett a. Den första beräkningen behandlar hela input-strängen och stannar i ett icke-accepterande tillstånd. Den tredje beräkningen behandlar också hela strängen men här stannar beräkningen i ett accepterande tillstånd vilket innebär att strängen accepteras.

En input-sträng accepteras om det finns en beräkning som behandlar hela strängen och stannar i ett accepterande tillstånd. För en **NFA** gäller precis samma definition som för en **DFA** nämligen följande definition

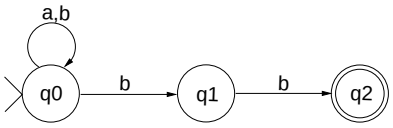
Definition 2.5 (Språket för en NFA)

Låt $M = (Q, \Sigma, \delta, S, F)$ vara en **NFA**. *Språket* för M, betecknat L(M), består av alla strängar u sådana $[S, u] \vdash^* [q, \varepsilon]$ för något $q \in F$.

En sträng tillhör språket för en **NFA** om det finns en beräkning som accepterar den; att det finns andra beräkningar som inte accepterar strängen är irrelevant. Den tredje beräkningen i exemplet ovan visar att strängen *ababb* tillhör språket för automaten. Man kan även observera att definitionen för språket för en NFA är identisk med definitionen för språket för en DFA. De två automaterna skiljer sig alltså inte på den punkten.

En **NFA** kan på motsvarande sätt som för en **DFA** visas grafiskt med hjälp av ett tillståndsdiagram. Faktum är att en **NFA** ritas på exakt samma sätt som en **DFA** där pil markerar starttillstånd och dubbla ringar betecknar accepterande tillstånd. Till skillnad mot tillståndsdiagrammet för en **DFA** kan nu, för varje tillstånd $q \in Q$ och input-tecken $a \in \Sigma$, finnas noll, en eller flera transitioner som utgår från q och är märkt a.

Exempel:
Tillståndsdiagrammet för exemplet ovan ser ut enligt följande:



Språket som automaten accepterar kan ur diagrammet uläsas till $(\boldsymbol{a} \cup \boldsymbol{b})^* \boldsymbol{bb}$.

ε-transitioner

De transitioner som hittills påträffats innebär att för att kunna byta tillstånd i en automat, såväl **NFA** som **DFA**, måste ett tecken från input-strängen läsas. Det finns dock vissa typer av **NFA** som tillåter transitioner utan att något input-tecken läses. Dessa transitioner kallas *epsilon-transitioner* och denna typ av automat kallas för **NFA**-ε.

Definition 2.6 (NFA-ε)

En **NFA**-ε definieras precis som en **NFA** förutom δ som definieras enligt följande: $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow Q$

Den kanske största användningen för ε-transitioner är i syfte att utifrån enklare automater bygga mer komplexa. Med fördel används ε-transitioner då reguljära uttryck omvandlas till automater. Det bör nämnas att en **NFA**-ε *inte* är kraftfullare än en vanlig **NFA** eller en **DFA** för den delen. Däremot kan ε-transitioner göra tillståndsdiagrammen för automaterna mer lättförståeliga och överskådliga. För faktum är att en **NFA**-ε enkelt kan transformeras till en **NFA** genom att ta bort ε-transitionerna. Denna **NFA** kan sedan, med algoritmer som kommer att presenteras senare, konverteras till en **DFA**. Det innebär, att alla dessa tre typer av ändliga automater är lika kraftfulla; det enda som skiljer är sättet de presenteras på. Detta kan sammanfattas av följande teorem:

Teorem 2.7 Låt L vara ett språk. Följande påståenden är ekvivalenta:

1. $L = L(A)$ där A är en **DFA**.
2. $L = L(A)$ där A är en **NFA**.
3. $L = L(A)$ där A är en **NFA**-ε.

Satsens giltighet bevisas inte här utan kommer visas på ett mer praktiskt sätt då algoritmer för konvertering mellan de olika typerna tas fram och implementeras.

3 Algoritmer för konvertering av automater

Den första av de fyra funktionerna är konvertering mellan olika typer av ändliga automater.

3.1 Introduktion

Hittills har tre olika typer av ändliga automater stötts på; **DFA**, **NFA** samt **NFA**-ε. De har dock alla sina bra och sina dåliga sidor. En **DFA** till exempel ger för varje input-sträng upphov till exakt en beräkning som enkelt kan avläsas i tillståndsdiagrammet genom att följa motsvarande väg i grafen. Man kan alltså omedelbart avgöra huruvida strängen ingår i automatens språk eller ej. Denna egenskap har inte en **NFA** som kanske måste prova olika vägar för att avgöra om strängen accpeteras. Med sina icke-deterministiska egenskaper behöver dock kanske inte en **NFA** lika många tillstånd och transitioner som en **DFA**; man måste komma ihåg att en **DFA** behöver en transition från varje tillstånd $q \in Q$ märkt a för alla $a \in \Sigma$. Detta kan bli många transitioner och en **NFA** kan bli lite mer överskådlig. Som senare kommer att visas kan en **NFA** med n tillstånd i värsta fallet kräva 2^n tillstånd för en likvärdig **DFA**. Till sist finns **NFA**-ε som har förmågan att "hoppa vidare" till ett annat tillstånd *utan* att konsumera något input-tecken. Denna egenskap kommer visa sig praktisk när man önskar skapa en automat utifrån ett reguljärt uttryck.

Som visats i 2.7 är alla dessa tre typer likvärdiga, det vill säga de kan alla tre representera samma språk. Borde då inte rimligtvis det finnas något sätt att konverta från den ena typen till nästa? Svaret är "jo", men hur går det till?

Konverteringsordning

För att kunna konvertera automaterna är det bra att veta i vilken ordning de skall konverteras och detta görs genom att analysera dem. En **NFA**-ε är den "mest generella" eftersom den inte bara accepterar Σ som input utan även ε. Inte helt oväntat finns möjligheten att ta bort ε-transitionerna och därmed skapa en helt likvärdig **NFA** utan ε-transitioner. Inte heller helt oväntat finns möjligheten att från en **DFA** skapa en **NFA** genom att undersöka kombinationer av tillstånd från **DFA**n.

1. Icke-deterministisk automat med ε-transitioner konverteras till
2. Icke-deterministisk automat utan ε-transitioner som konverteras till
3. Deterministisk automat.

Varför konvertera automater?

En nyfiken läsare undrar nu givietvis; varför konvertera automaterna överhuvudtaget? Svaret på den frågan är rätt och slätt: avgörbarhet. Till saken hör nämligen att ett reguljärt språk, må det vara representerat av en automat, ett reguljärt uttryck eller en reguljär grammatik representeras *alltid* av en och samma deterministiska ändliga automat som minimerats[2]. Det betyder att det är möjligt att avgöra om två automater accepterar samma språk; genom att konvertera till **DFA** och sedan minimera och jämföra automaterna. Är språken lika kommer de minimerade automaterna att vara identiska. Denna egenskap är just det som gör reguljära språk (och i synnerhet ändliga automater) perfekta att lära sig det abstrakta tankesätt som är formella språk. Som tidigare nämnts har PDA (pushdown-automater), som är en kraftfullare automat och representerar de kontext-fria språken, inte denna egenskap och det är bevisligen omöjligt avgöra huruvida två automater representerar samma språk.

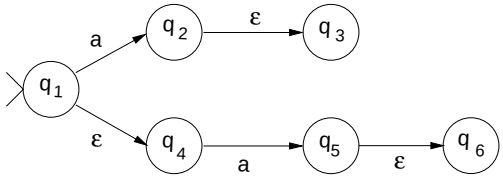
Med ovanstående information om minimering kan följande läggas till konverteringsordningen:

4. Minimera deterministisk automat.

3.2 ε-transitioner eller inte?

Det allra första steget i omvandling av en **NFA** till en **DFA** är att ta bort ε-transitioner. För att göra detta introduceras en modifierad transitionsfunktion t, på engelska kallad *input transition function*, som är den mängd tillstånd som kan nås från ett givet tillstånd genom att läsa *ett* input-tecken. Skillnaden mellan den vanliga transitionsfunktionen δ och t är att den nya funktionen inte innehåller några ε-transitioner, detta eftersom vid denna typ av transitioner konsumeras inga input-tecken. För att förstå det hela bättre kan man betrakta följande figur:

Väg	Sträng inläst
q_1, q_2	a
q_1, q_2, q_3	a
q_1, q_4	ε
q_1, q_4, q_5	a
q_1, q_4, q_5, q_6	a



Det syns nu tydligt att från tillståndet q_1 kan man komma till tillstånden $\{q_2, q_3, q_5, q_6\}$ efter att bara ha konsumerat ett a från input-strängen. Observera även att q_4 inte finns med eftersom transitionen till q_4 inte kräver något konsumerat tecken.

Definitionen av mängden *t(q_i, a)* kan delas in i tre delar; först skapas en mängd som innehåller alla tillstånd som kan nås med ε-transitioner från q_i. Från alla dessa tillstånd konsumeras ett a och från de tillstånd som nås via dessa transitioner följs ε-transitioner på nytt och mängden *t(q_i, a)* är klar. För att underlätta detta definieras ett nytt begrepp *ε-closure*.

[2]En minimerad automat är en automat som innehåller så få tillstånd som är möjligt.

Definition 3.1 (ε-closure)

ε-closure för ett tillstånd q_i, betecknat *ε-cl(q)* definieras rekursivt enligt:

(1) Bas: $q \in$*ε-cl(q)*.

(2) Rekursivt steg: Låt q' vara ett element i *ε-cl(q)*. Om $q'' \in \delta(q',\varepsilon)$ så $q'' \in$ *ε-cl(q)*.

(3) Slutsats: q' tillhör *ε-cl(q_i)* enbart om det kan erhållas från q genom ett ändligt antal tillämpningar av det rekursiva steget.

Vad definitioner säger är att ε-closure för ett tillstånd är alla de tillstånd som kan nås via ε-transitioner. För exemplet ovan är ε-closure för $q_1 = \{q_1, q_4\}$. Hade en ε-transition funnits mellan q_4 och q_5 skulle således även q_5 varit med i ε-closure för q_1, detta trots att q_1 och q_5 inte där direkta grannar; det rekursiva steget i definitionen ovan gör det möjligt att i flera steg finna tillstånd till ε-closure.

Nu används ovanstående definition av ε-closure för att definiera *input transition function*.

Definition 3.2 (Input transition function)

Input transition function t hos en automat definieras enligt:

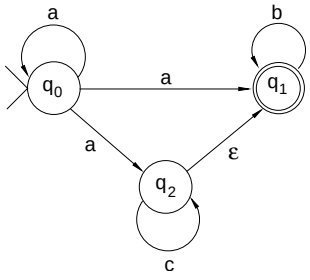
$$t(q,a) = \bigcup_{q' \in \varepsilon - cl(q)} \varepsilon - cl(\delta(q',a))$$

för alla $q \in Q$ och $a \in \Sigma$.

Informellt kan funktionen uttryckas enligt följande: "Från varje tillstånd q' i *ε-cl(q)* görs en transition via input-tecknet a. Unionen av ε-closure för de tillstånd som nås via de tidigare givna transitionerna utgör t för tillståndet q och input-tecken a."

Att finna *input transition function* för en automat är förmodligen enklare än att förstå den formella definitionen av den. För att underlätta förståelsen visas det med ett exempel:

Exempel:
Antag att man har en **NFA**-ε som accepterar språket $\boldsymbol{a^+ c^* b^*}$.



δ	a	b	c	ε
q_0	$\{q_0, q_1, q_2\}$	$\emptyset$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1\}$

t	a	b	c
q_0	$\{q_0, q_1, q_2\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$	$\{q_1, q_2\}$

Skillnaden mellan t och δ är som synes att alla ε-transitioner har försvunnit, detta tack vare att definitionen ovan utnyttjar det faktum att alla ε-transitioner är "borträknade". Den önskade **NFA** utan ε-transitioner är nu erhållen; dess transitionsfunktion δ är helt sonika input transition function t enligt ovan.

Sluttillstånd är alla tidigare sluttillstånd samt, ifall ett sluttillstånd finns i dess ε-closure, även starttillståndet.

Implementera *Input transition function*

För att kunna implementera en input transition function enligt definitionen ovan behövs en algoritm som på något sätt är möjlig att implementera. Innan en sådan algoritm är möjlig behövs självklart också ε-closure implementeras. Detta görs enklast genom att följa definitionen för ε-closure enligt ovan. En tänkbar implementering ser ut enligt följande pseudokod:

```
for (i = 0; i < card(Q); i++) {
  e-cl(Q(i)) = { Q(i) }
  while (e-cl(Q(i))->current != NULL) { /* Otestade tillstånd kvar */
    q = e-cl(Q(i))->current
    for (k = 0; k < card(Q); k++) {
      if (Q(k) in d(q, e)
          add (Q(k), e-cl(Q(i)))
    }
    e-cl(Q(i))->current = e-cl(Q(i))->next
  }
}
```

Koden fungerar enligt följande: `Q` är mängden av alla tillstånd i automaten. Låt `Q(i)` vara tillstånd i av dessa. `Q(i)` kommer alltid finnas i ε-closure för `Q(i)`. För alla tillstånd i ε-closure för `Q(i)` där `q` betecknar aktuellt tillstånd av dessa, gå igenom samtliga tillstånd i `Q` och om k:te tillståndet `Q(k)` $\in \delta(q, \varepsilon)$ så lägg till `Q(k)` till ε-closure för `Q(i)`. `card(Q)` betecknar kardinaliteten för Q, det vill säga antalet tillstånd som automaten har. Man bör observera att ε-closure för `Q(i)` hela tiden utökas varför denna bäst implementeras med en länkad lista. Därför används en `while`-slinga, som är mer dynamisk än exempelvis en `for`-loop.

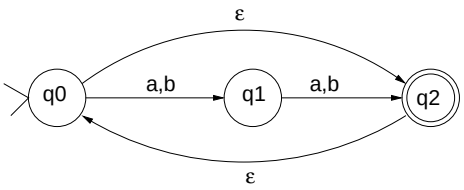
Nästa steg är att implementera själva input transition function och utifrån denna skapa den nya **NFA** utan ε-transitioner. För detta ändamål kan en algoritm enligt följande pseudokod implementeras:

```
for (i = 0; i < card(Q); i++) {
  for (j = 0; j < card(S); j++) {
  union = empty();
    for (k = 0; k < card(e-cl(Q(i))); k++) {
      q = get_ecl(Q(i), k)    /* Tillstånd k ur e-cl(Q(i) */
      for (l = 0; l < card(Q); l++) {
        if (Q(l) in d(q, S(j))
          add(e-cl(Q(l)), union)
      }
    }
    t(Q(i), S(j)) = union;
  }
}
```

Denna kod fungerar så här: `Q` är mängden av alla tillstånd Q i automaten och `S` betecknar mängden av alla input-tecken, dvs Σ. Låt `q` vara tillstånd k ur ε-closure för `Q(i)` och för input-tecken `S(j)` i Σ, gå igenom alla tillstånd i `Q` och om det l:te tillståndet `Q(l)` av dessa återfinns i $\delta(q, S(j))$, lägg ε-closure för $Q(l)$ till en specifik union. Denna union kommer till slut att bli t för tillstånd $Q(i)$ och input-tecken $S(j)$.

Exempel:

Betrakta följande automat M:



Denna automat har en transitionsfunktion δ som ser ut enligt följande:

δ	a	b	ε
q_0	$\{q_1\}$	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_0\}$

Först behövs ε-closure för alla tillstånd i M. Dessa kan erhållas enligt algoritmen ovan men det går att ur figuren utläsa att dessa är $\{q_0, q_2\}, \{q_1\}, \{q_0, q_2\}$ för q_0, q_1 respektive q_2. Nu används andra algoritmen ovan för att erhålla input transition function t:

q_0:
ε-closure för $q_0 = \{q_0, q_2\}$
För a: $\delta(q_0, a) = \{q_1\}$. ε-closure$(q_1) = \{q_1\}$. Lägg till $\{q_1\}$ till unionen.
$\delta(q_2, a) = \emptyset$. Inget mer att göra.
Klart för a: unionen består av $\{q_1\}$.

För b: $\delta(q_0, b) = \{q_1\}$. Samma som ovan; lägg till $\{q_1\}$ till unionen
$\delta(q_2, b) = \emptyset$. Inget mer att göra.
Klart för b: unionen består av $\{q_1\}$.

q_1:
ε-closure för $q_1 = \{q_1\}$
För a: $\delta(q_1, a) = \{q_2\}$. ε-closure$(q_2) = \{q_0, q_2\}$. Lägg till $\{q_0, q_2\}$ till unionen.
Klart för a: unionen består av $\{q_0, q_2\}$.

För b: $\delta(q_1, b) = \{q_2\}$. Samma som ovan.
Klart för b: unionen består av $\{q_0, q_2\}$.

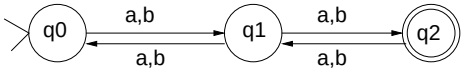
q_2:
ε-closure för $q_2 = \{q_0, q_2\}$
För a: $\delta(q_0, a) = \{q_1\}$. ε-closure$(q_1) = \{q_1\}$. Lägg till $\{q_1\}$ till unionen.
$\delta(q_2, a) = \emptyset$. Inget mer att göra.
Klart för a: unionen består av $\{q_1\}$.

För b: $\delta(q_0, b) = \{q_1\}$. Samma som ovan; lägg till $\{q_1\}$ till unionen
$\delta(q_2, b) = \emptyset$. Inget mer att göra.
Klart för b: unionen består av $\{q_1\}$.

Algoritmen är nu klar och input transition function t har erhållits och ser ut enligt följande:

t	a	b
q_0	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_0, q_2\}$	$\{q_0, q_2\}$
q_2	$\{q_1\}$	$\{q_1\}$

Tillståndsdiagrammet för automaten ser ut så här:



3.3 Konvertera en NFA till en DFA

Nästa steg i konverteringsprocessen är att skapa en deterministisk automat utifrån en icke-deterministisk. Detta innebär alltså att på något sätt "få bort" automatens icke-deterministiska egenskaper. Hur görs då detta? Enkelt uttryckt kan man säga att en deterministisk automat besöker *alla* möjliga tillstånd i

en icke-deterministisk automat *samtidigt*. Med andra ord är tillstånden i den deterministiska automaten mängder av tillstånd i den icke-deterministiska.

Det allra vanligaste sättet att konvertera en **NFA** M till en **DFA** DM går till enligt följande algoritm.

Input: en **NFA** $M = (Q, \Sigma, \delta, S, F)$.
Output: en **DFA** $DM = (Q', \Sigma, \delta', \{S\}, F')$ som representerar samma språk.

1 Initialt, sätt Q' till S.

2 **Repetera**

2.1 Om det finns ett tillstånd $X \in Q'$ och ett input-tecken $a \in \Sigma$ utan transition från X märkt a så:

2.1.1 Låt Y vara unionen av $\delta(q_i, a)$ för alla $q_i \in X$.

2.1.2 Om $Y \notin Q'$ så låt $Q' = Q' \cup \{Y\}$, dvs lägg till Y till Q.

2.1.3 Lägg till en transition från X till Y märkt a.

Annars **klar**.

tills **klar**

3 Sätt accepternade tillstånd F' i DM till $F' = \{X \in Q' \mid X \cap F \neq \emptyset\}$

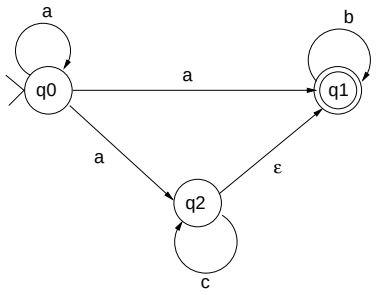
Algoritmen innebär kortfattat att för varje tillstånd i nya mängden Q' kontrollerar man huruvida den har en transition för varje input-tecken. Om så ej är fallet bildar man en mängd av tillstånd från Q Man följer alltså *varje* transition från det aktuella tillståndet och lägger samman resultaten. Om detta nya tillstånd (denna union av gamla tillstånd) inte existerar i Q' läggs det till.

Implementation av konvertering från NFA till DFA

Eftersom konvertering från en **NFA** till en **DFA** följer en väldigt klar och tydlig algoritm bör implementering av denna inte skapa några större bekymmer. Frågan är däremot hur det skall lösas rent praktiskt. En lösning på det hela är att utnyttja det faktum att vad det hela handlar om är att slå ihop tillstånd från en ursprunglig **NFA** för att skapa nya tillstånd i en **DFA** och i och göra detta genom att skapa en länkad lista där varje element är en lista i sig innehållandes just de gamla tillstånden. Listan fylls på efterhand med fler och fler grupper av tillstånd (som alltså bildar de nya tillstånden) och genom att sortera varje grupp och utifrån de ingående tillståndens nummer skapa etiketter är det väldigt enkelt att hålla koll på om en grupp av tillstånd är ny eller en redan existerande (detta i och med att varje tillstånd måste ha en unik etikett; varje tillstånd lagras med ett unikt nummer men för läsbarhetens skull - för att inte tala om pedagogikens - måste de ha unika etiketter så att användaren enkelt kan skilja dem åt). När inte längre några nya tillstånd skapas är automaten klar och man finner enkelt accepterande tillstånd genom att gå igenom listan med originaltillstånden; alla de grupper (tillstånd) som innehåller accepterande tillstånd ur gamla automaten, alltså element ur F i ovanstående algoritm, kommer att vara element i F', det vill säga accepterande tillstånd i den nya automaten.

Exempel:

Följande exempel visar, genom att följa varje steg i algoritmen, hur det går till (och visar samtidigt att algoritmen verkligen fungerar). Antag att följande **NFA**-ε ska konverteras:



Genom att använda algoritmern från förra kapitlet erhålls input transition function t (vilket motsvarar den transitionsfunktion δ som en **NFA** utan ε-transitioner skulle ha). Den flitige (eller för den delen, osäkre) läsaren kan kontrollera att denna funktion t stämmer.

t	a	b	c
q_0	$\{q_0, q_1, q_2\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$
q_2	$\emptyset$	$\{q_1\}$	$\{q_1, q_2\}$

Nu påbörjas algoritmen enligt ovan:

1. Initiera Q' till $\{\{q_0\}\}$.
2. Repetera:

2.1 Det finns ett tillstånd i Q' som inte har någon kant märkt a. Kalla detta tillstånd $X = \{q_0\}$.

2.1.1 Låt $Y = t(q_0, a) = \{q_0, q_1, q_2\}$.

2.1.2 $Y \notin Q' \rightarrow Q' = Q' \cup Y$ dvs lägg till Y till Q'.

2.1.3 $\delta(X, a) = Y$ dvs skapa en transition från X till Y märkt a.

2.1 Det finns ett tillstånd i Q' som inte har någon kant märkt b. Kalla detta tillstånd $X = \{q_0\}$.

2.1.1 Låt $Y = t(q_0, b) = \emptyset$

2.1.2 $Y \notin Q' \rightarrow Q' = Q' \cup Y$.

2.1.3 $\delta(X, b) = Y$.

2.1 Det finns ett tillstånd i Q' som inte har någon kant märkt c. Kalla detta tillstånd $X = \{q_0\}$.

2.1.1 Låt $Y = t(q_0, c) = \emptyset$

2.1.2 $Y \in Q' \rightarrow Q' = Q'$

2.1.3 $\delta(X, c) = Y$.

Nu har ett första steg av algoritmen gjorts och den nya automaten innehåller tre tillstånd: $\{\{q_0\}, \{q_0, q_1, q_2\}, \emptyset\}$. Samma procedur upprepas nu för nästa tillstånd, nämligen $\{q_0, q_1, q_2\}$ och efter detta steg har ytterligare två tillstånd, $\{q_1\}$ och $\{q_1, q_2\}$, skapats.

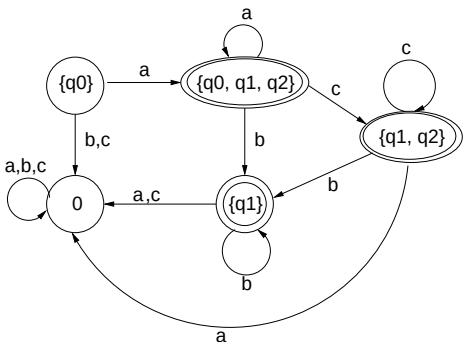
När samtliga dessa traverserats har inga fler tillstånd skapats och följande fem nya tillstånd har alltså skapats:

$$Q' = \{\{q_0\}, \{q_0, q_1, q_2\}, \emptyset, \{q_1\}, \{q_1, q_2\}\}$$

Slutligen, det sista steget i algoritmen; att finna accepterande tillstånd för den nya automaten DM.

3 F', mängden av alla accepterande tillstånd i den nya automaten är de tillstånd vilka innehåller något accepterande tillstånd ur F. Eftersom $F = \{q_1\}$ blir $F' = \{\{q_0, q_1, q_2\}, \{q_1\}, \{q_1, q_2\}\}$.

Resultatet av konverteringen blir följande DFA:



3.4 Minimera en deterministisk automat

Hittills har tre av de fyra konverteringsstegen som presenterades inledningsvis i detta kapitel undersökts och algoritmer för dessa samt förslag på hur de skall implementeras har redovisats. Dessutom har exempel på hur algoritmerna används visats på enkla, ändliga automater. Nu återstår endast ett steg kvar; minimering av en deterministisk automat.

Som visades i förra kapitlet kan en **NFA** alltid konverteras till en **DFA** och idén med algoritmen är att gruppera tillstånd i **NFA**n för att skapa nya, deterministiska tillstånd. Eftersom varje nytt tillstånd är en delmängd av Q kan inte

mindre än $card(\mathcal{P}(Q)) = 2^n$ nya tillstånd krävas. En **NFA** med 4 tillstånd kan alltså kräva upp till 16 tillstånd som motsvarande **DFA**. Ovanstående algoritm för konvertering mellan **NFA** och **DFA** kommer alltid att terminera eftersom maximalt $card(\mathcal{P}(Q)) \cdot card(\Sigma)$ iterationer behövs. Att de båda automaterna är ekvivalenta bevisas exempelvis på sid. 22 i [2] och sid. 69 i [3].

Eftersom antalet tillstånd i en **DFA** vida kan överstiga antalet för motsvarande **NFA** finns en risk att antalet tillstånd blir många och automaten blir överskådlig. Kan det då inte tänkas att några av dessa tillstånd på något sätt blir överflödiga? Kan inte två tillstånd tänkas vara likadana? Faktum är att så är fallet; detta kallas att tillstånden är *ekvivalenta* eller *oskiljaktliga* och dessa kan då slås ihop - automaten *minimeras*.

Ekvivalenta tillstånd

Minimering av automater är absolut nödvändigt för att kunna avgöra om två automater representerar samma språk och för att kunna minimera en automat måste mellan två tillstånd kunna slås fast. Vad är då egentligen ekvivalens? En definition på ekvivalens skulle vara på sin plats:

Definition 3.3 (Ekvivalenta tillstånd)

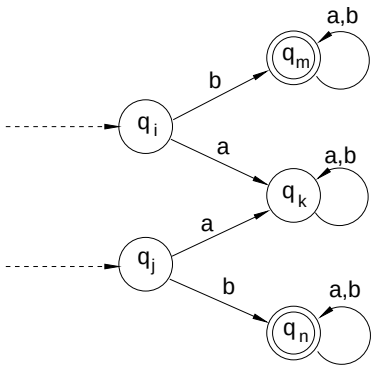
Två tillstånd q och q' anses vara ekvivalenta (eng: *indistinguishable*[3]) om

$$\hat{\delta}(q, u) \in F \Leftrightarrow \hat{\delta}(q', u) \in F$$

för alla strängar $u \in \Sigma^*$ och där $\hat{\delta}(q, u)$ innebär att *hela* strängen u konsumeras med början i q (det vill säga noll eller fler appliceringar av δ).

Vad definitionen säger är att om en sträng $u \in \Sigma^*$ kan påbörjas konsumeras i såväl q som q' och båda dessa leder till att strängen accepteras, då är tillstånden q och q' ekvivalenta och kan därmed slås ihop till ett enda tillstånd.

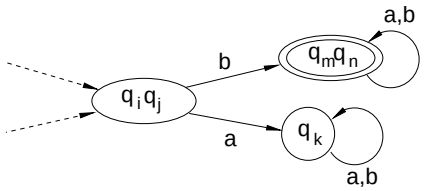
Ekvivalens mellan två tillstånd kan beaktas i följande figur:



[3] En direkt översättning av "indistinguishable" är "oskiljaktlig" men "ekvivalent" är ett bättre ord

De streckade linjerna som kommer till q_i och q_j indikerar att sättet på vilket tillstånden nås är irrelevant för den fortsatta beräkningen; det fastslogs redan tidigare att en ändlig automat inte är intresserad av tidigare konsumtion av input-tecken, bara aktuellt tillstånd samt kvarvarande tecken är intressant. En noggrann kontroll visar att tillstånden q_i och q_j är ekvivalenta. Varför? Jo, eftersom beräkningen av en sträng som börjar med konsumtion av b kommer att avslutas i ett accepterande tillstånd (eftersom beräkningen inte kommer vidare från q_m eller q_n) och beräkningen av andra strängar hamnar i icke-accepterande tillstånd, spelar det ingen roll om beräkningen hamnar i q_i eller q_j; slutresultatet är detsamma och tillstånden kan därför anses vara ekvivalenta. Samma sak gäller tillstånden q_m och q_n; beräkningar som hamnar i dessa tillstånd kommer att sluta i ett accepterande tillstånd oavsett vilken väg de tidigare tagit för att hamna där. De är därför också ekvivalenta.

Ekvivalenta tillstånd kan som tidigare sagts slås ihop vilket gör att sammanslagningar av tillstånd i figuren ovan ger istället följandet figur:



Minimeringsprocessen

Det finns flera olika metoder för att minimera en deterministisk ändlig automat. I denna rapport kommer två olika metoder att visas; den ena är en mer teoretisk metod som är enkel att förklara men svårare att implementera, den andra är precis tvärtom - svår att förklara men enkel att implementera. Att båda två ändå visas och förklaras beror på att den första visar på ett förståeligt sätt principen bakom minimering.

Den metod som implementeras, som återfinns i till exempel [4], innebär att man väljer att identifiera ekvivalenta tillstånd genom att med varje par av tillstånd (q_i, q_j), $i < j$ associera två värden $D[i,j]$ och $S[i,j]$ där $D[i,j] = 1$ anger att q_i och q_j ej är ekvivalenta och mängden $S[i,j]$ innehåller par av index av tillstånd som anger hur q_i, q_j kan skiljas åt.

Den ovan beskrivna metoden är inte speciellt intuitiv och istället brukar en alternativ metod användas. Denna finns beskriven till exempel på sid. 141 i [1]. Denna metod går ut på att dela in tillstånden i olika mängder och därefter kontrollera huruvida tillstånden i mängden uppför sig på liknande sätt.
Antag att automaten $DM = (Q, \Sigma, \delta, S, F)$ skall minimeras. Följande algoritm kan användas i detta syfte.

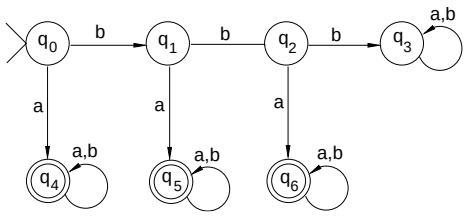
1. Initialt skapas två mängder X_1 och X_2 där $X_1 = Q \setminus F$ och $X_2 = F$ det vill säga den ena mängden innehåller alla accepterande tillstånd och den andra innehåller alla övriga tillstånd.

2. För varje tillstånd $q_i \in X_1$, kontrollera huruvida $\delta(q_j, a) \notin X_1$ för alla $a \in \Sigma$, det vill säga: finns det något tillstånd som har en transition som leder till ett tillstånd utanför X_1? I så fall, skapa två nya mängder: $X_1 = X_1 \setminus q_i$ och $X_3 = q_i$.

3. Upprepa proceduren för X_i för alla i, det vill säga fortsätt att skapa nya mängder innehållande avvikande tillstånd. När inga nya mängder kan skapas är algoritmen färdig och alla tillstånd inom samma mängd är ekvivalenta och kan slås ihop till ett tillstånd.

Exempel:

Antag att följande **DFA** M skall minimeras:



Enligt algoritmens första steg skapar man två mängder: $X_1 = \{q_0, q_1, q_2, q_3\}$ och $X_2 = \{q_4, q_5, q_6\}$. Första elementet i X_1 är q_0. $\delta(q_0, a) = q_4$ och eftersom q_4 inte återfinns i X_1 skapar man två nya tillstånd genom att lyfta ut q_0 ur X_1 och lägga detta i ett eget tillstånd, X_3.

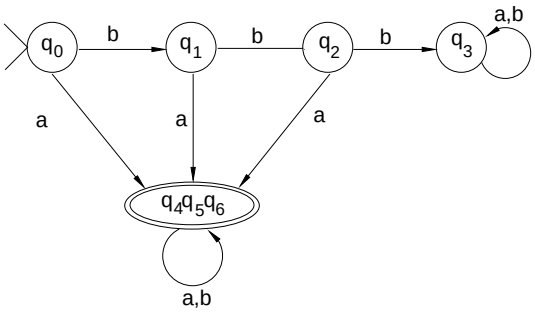
Sedan fortsätter man med nästa tillstånd i X_1 som är q_1. $\delta(q_1, b) = q_5$ och eftersom inte heller q_5 återfinns i X_1 lyfts q_1 ur X_1 och placeras i X_4.

Nästa tillstånd i X_1 är q_2, som har en transition till q_6 som är utanför X_1. Därför skapas ytterligare en mängd X_5 innehållande tillståndet q_2. Nu återstår endast ett tillstånd i X_1, varför man inte behöver fortsätta. En titt på tillstånden visar att efter att X_1 behandlats ser de ut enligt följande:

$$X_1 = \{q_3\},\ X_2 = \{q_4, q_5, q_6\},\ X_3 = \{q_0\},\ X_4 = \{q_1\},\ X_5 = \{q_2\}$$

Nästa steg är att behandla tillstånden i X_2. Man ser dock att för vart och ett av tillstånden i X_2 kommer inga transitioner att lämna X_2 (faktum är ju att alla transitioner för varje tillstånd leder tillbaka till tillståndet självt) vilket leder till att inga ytterligare uppdelningar kan göras. Eftersom mängderna X_3, X_4 och X_5 bara innehåller ett element var kan dessa inte delas upp ytterligare.

Minimeringen är nu klar, inga fler uppdelningar kan längre göras och det som återstår nu är att slå ihop de tillstånd som finns i varje mängd. Eftersom X_2 innehåller fler än ett tillstånd slår man ihop q_4, q_5 och q_6 till ett enda tillstånd och automaten ser efter minimering ut enligt följande:



Implementering av minimeringsalgoritm

Nu kan man ju ställa sig frågan; varför går det inte att implementera ovanstående algoritm? Svaret är att det går att implementera men det är inte lätt; speciellt blir det svårt när antalet tillstånd ökar. Vad är då alternativet? Jo, det är att återgå till det som nämndes först, nämligen att införa två variabler, D och S. Dessa två, tillsammans med en rekursiv funktion används för att skapa en algoritm som parvis av alla möjliga tillstånd undersöker om de två tillstånden är ekvivalenta eller ej och som dessutom sparar information för senare användning.

Den minimeringsalgoritm som implementeras, som hämtas direkt från [4] sid. 184, ger en relativt enkelt implementation eftersom den är väldigt rakt på sak och med hjälp av lite förklaring ändå ganska enkelt att förstå. Algoritmen ser ut enligt följande:

1 Initiering

for varje par av tillstånd q_i och $q_j, i < j$ **do**

1.1 $D[i,j] = 0$

1.1 $S[i,j] = \emptyset$

end for

2. **for** varje par $i,j; i < j$ **if** antingen q_i eller q_j är ett accepterande tillstånd och det andra inte, **then** sätt $D[i,j] = 1$.

3. **for** varje par $i,j; i < j$ med $D[i,j] = 0$ **do**

3.1 **if** för något $a \in \Sigma, \delta(q_i,a) = q_m$ och $\delta(q_j,a) = q_n$ och $D[m,n] = 1$ or $D[n,m] = 1$ **then** $DIST(i,j)$

3.2 **else** för varje $a \in \Sigma$ **do**: Låt $\delta(q_i, a) = q_m$ och $\delta(q_j, a) = q_n$
if $m < n$ och $[i, j] \neq [m, n]$, **then** lägg till $[i, j]$ till $S[m, n]$
else if $m > n$ och $[i, j] \neq [n, m]$ **then** lägg till $[i, j]$ till $S[n, m]$

end for

$DIST(i, j)$;

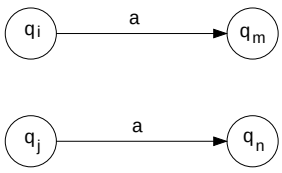
begin

$D[i, j] = 1$

for alla $[m, n] \in S[i, j]$, $DIST(m, n)$

end

För att förstå algoritmen kan följande bild beaktas:



Om q_m och q_n satts som icke-ekvivalenta när q_i och q_j undersöks i steg 3 så sätts $D[i, j]$ till 1 för att indikera att q_i och q_j också de är icke-ekvivalenta; detta i och med att de via transition för samma input-tecken leder fram till två icke-ekvivalenta tillstånd och därmed inte kan vara ekvivalenta. Om det inbördes förhållandet mellan q_m och q_n skulle vara okänt då q_i och q_j undersöks men en senare undersökning visar att de *är* icke-ekvivalenta, ja då kommer också q_i och q_j att vara icke-ekvivalenta; denna information sparas i variabeln S på sådant sätt att $[i, j] \in S[m, n]$ vilket kan översättas med att icke-ekvivalens mellan q_m och q_n räcker för att påvisa icke-ekvivalens mellan q_i och q_j. Denna del sköts i algoritmen av den rekursiva funktionen $DIST$. För att bestämma vilka tillstånd som är ekvivalenta tar man helt enkelt bara de par av index för vilka $D = 0$. Dessa tillstånd kan nu slås ihop till ett. Skulle ett tillstånd vara ekvivalent med ett tredje kan sedan detta slås ihop med de tidigare två och så vidare (då visar det sig självklart att tillstånd två och tre också är ekvivalenta, men det är ju trivialt).

Exempel

För att testa att algoritmen fungerar vore ett exempel på sin plats och varför då inte testa den automat som visas ovan? I steg två, och direkt ur automaten, ges att 1 tilldelas till följande värden: $D[0,4]$, $D[0,5]$, $D[0,6]$, $D[1,4]$, $D[1,5]$, $D[1,6]$, $D[2,4]$, $D[2,5]$, $D[2,6]$, $D[3,4]$, $D[3,5]$ samt $D[3,6]$. Steg tre i algoritmen kommer att ge följande:

Index	**Händelse**	**Orsak**
$[0,1]$	$S[4,5] = \{[0,1]\}$	
	$S[1,2] = \{[0,1]\}$	
$[0,2]$	$S[4,6] = \{[0,2]\}$	
	$S[1,3] = \{[0,2]\}$	
$[0,3]$	$D[0,3] = 1$	Skiljer sig m.a.p. a
$[1,2]$	$S[5,6] = \{[1,2]\}$	
	$S[2,3] = \{[1,2]\}$	
$[1,3]$	$D[1,3] = 1$	Skiljer sig m.a.p. a
	$D[0,2] = 1$	Anrop av DIST(1, 3)
$[2,3]$	$D[2,3] = 1$	Skiljer sig m.a.p. a
	$D[1,2] = 1$	Anrop av DIST(2, 3)
	$D[0,1] = 1$	Anrop av DIST(1, 2)
$[4,5]$		
$[4,6]$		
$[5,6]$		

Algoritmen stannar och det är uppenbart att tre tillstånd kan slås ihop till ett; q_4, q_5 och q_6, vilket också var slutledningen som dragits tidigare. Detta visar alltså att algoritmen fungerar fint och torde därför kunna implementeras utan större besvär.

I och med slutförandet av minimeringsalgoritmen har nu fyra olika algoritmer tagits fram; var och en betecknar ett steg i konverteringen från en **NFA**-ε till en minimerad **DFA**. I nästa kapitel kommer **NFA**-ε att utnyttjas mer för då kommer algoritmer för att skapa ändliga automater utifrån reguljära uttryck att skapas.

4 Algoritmer för att skapa ändliga automater från reguljära uttryck

Det har tidigare visats att ett reguljärt språk kan representeras på tre olika sätt; som en ändlig automat, som ett reguljärt uttryck och som en reguljär grammatik, av vilka den sista helt åsidosätts i detta arbete.

Finns det då inte någon koppling mellan en ändlig automat och ett reguljärt uttryck? Visst är det så, men faktum är att en ändlig automat och ett reguljärt uttryck kan betraktas som varsin sida av myntet; å ena sidan finns en ändlig automat som kan avgöra huruvida en sträng är med i språket eller ej och å andra sidan finns ett reguljärt uttryck som gör något som en ändlig automat *aldrig* kan göra, nämligen direkt tala om *vilka* strängar som språket består av. Automaten kan alltså betraktas som en *språkaccepterare* medan ett reguljärt uttryck är en *språkgenererare*.

I detta kapitel visas hur, med mycket enkla medel, en ändlig automat kan skapas utifrån ett reguljärt uttryck samt hur en sådan algoritm skall implementeras. Det finns tre huvudsakliga operationer på reguljära uttryck:

- Konkatenering; Sammanslagning av två uttryck.
- Union; val mellan två olika uttryck.
- Kleene star; upprepning av ett uttryck noll eller fler gånger.

Utöver dessa finns en del andra operationer av vilka +-operatorn kan nämnas. Den betecknar en upprepning av ett uttryck *en* eller fler gånger, det vill säga en konkatenering av ett uttryck och samma uttryck med Kleene star-operatorn. Anledningen till att denna operator nämns är att den är, efter att konkatenering och Kleene star implementerats, är ytterst enkel att implementera; den använder ju som sagt de tidigare nämnda operatorerna.

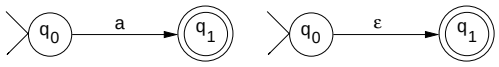
För att nämna något om att gå åt andra hållet, det vill säga skapa ett reguljärt uttryck utifrån en automat, kan nämnas att det är en aning svårare vilket kräver en djupare teoretisk bakgrund och en hel del programmering för att implementera. Detta har därför inte tagits upp. Algoritm för detta finns exempelvis på s.200 i [4].

4.1 Konvertering av reguljära uttryck

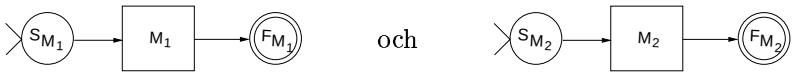
Själva metoden för att skapa automater utifrån reguljära uttryck är en algoritm som är väldigt enkel och rättfram. Den bygger på användandet av två generella, atomära uttryck eller automater som är grundstenarna. Dessa används sedan för att med hjälp av ε-transitioner skapa nya, större automater vilka i sig sedan kan byggas vidare på.

Grundläggande automater

De enklaste av automater som kan skapas är de automater som accepterar ett input-tecken eller ε. Dessa skapas så här:



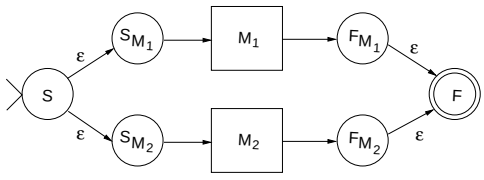
Istället för att använda specifika automater som accepterar ett speciellt språk kommer i detta kapitel istället två generella automater att användas.



Dessa automater accepterar språken $L(M_1)$ respektive $L(M_2)$. Vad som döljer sig inuti de vita boxarna är oväsenligt; huvudsaken är att automaterna accepterar de ovan angivna språken.

Unioner

Unionen av två automater M_1 och M_2 kan enkelt skapas genom att först skapa ett nytt starttillstånd S som är helt fristående från de övriga. Från detta nya starttillstånd utgår sedan ε-transitioner till starttillstånden för de gamla automaterna som ju accepterar språken $L(M_1)$ och $L(M_2)$. Från de accepterande tillstånden i de tidigare automaterna skapas sedan ε-transitioner till ett nytt, generellt accepterande tillstånd. Den nya automaten kommer härvid att acceptera $L(M_1) \cup L(M_2)$ vilket är syftet med en union; att skapa ett val mellan antingen den ena eller den andra automaten. Observeras bör att starttillstånden och accepterande tillstånden i automaterna M_1 och M_2 förlorar sina egenskaper; detta eftersom nytt start- respektive sluttillstånd har skapats.

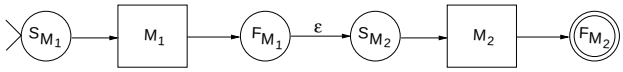


Konstruktionen kan ses som att automaterna M_1 och M_2 "körs parallellt" och att en sträng som testas av en automat accepteras om den accepteras av någon av de två automaterna, det vill säga någon av de två vägarna i grafen.

Konkatenering

Till skillnad mot union där automaterna "körs parallellt" kan man se konkateneringen av M_1 och M_2 som att de "körs seriellt" istället. Starttillstånd är

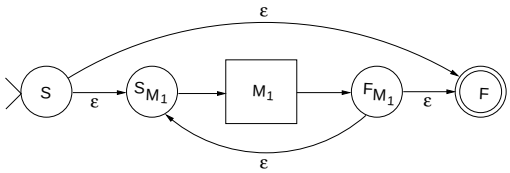
startillståndet för M_1 och accepterande tillstånd blir accepterande tilltånd för M_2. Automaterna slås ihop med en ε-transition mellan accepterande tillstånd i M_1 och starttillstånd i M_2, vilka självklart förlorar sina egenskaper. Automaten får då följande utseende.



För att en sträng som testas av automaten skall accepteras måste den först ta sig igenom M_1 och därefter måste resterande del av strängen accepteras av M_2. Då har strängen accepteras av automaten M_1M_2.

Kleene star

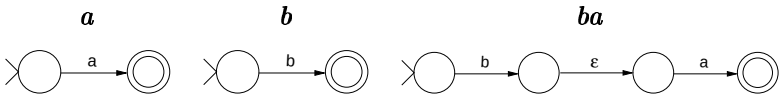
En automat som accepterar $L(M_1)^*$ måste kunna passera genom M_1 ett godtyckligt antal gånger. Detta görs med en ε-transition från det accepterande tillståndet till starttillståndet. För att uppnå målet med Kleene star (*noll* eller fler) behövs även en ε-transition från starttillståndet till det accepterande tillståndet. Detta gör att en sträng kan passera M_1 utan att konsumera något tecken. Observera ett nytt starttillstånd och ett nytt sluttillstånd har skapats. ε-transitioner från de nya till de gamla start- respektive sluttillstånden skapas också.



Att skapa en automat för +-operatorn är också enkelt och det finns två sätt; antingen skapa en konkatenering mellan M_1 och M_1^* eller ta bort ε-transitionen från starttillståndet till det accepterande tillståndet i automaten för M_1^* ovan.

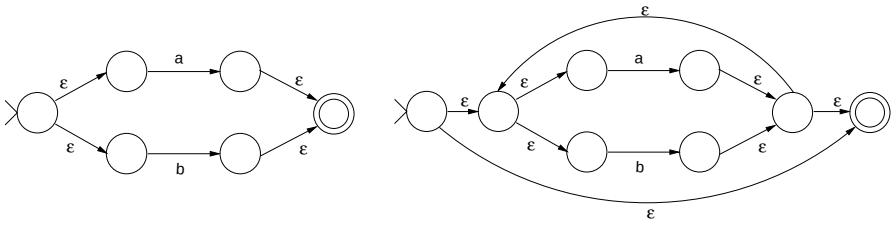
Ett enkelt exempel

De tekniker som presenterats ovan kommer att i följande exempel användas för att utifrån det reguljära uttrycket $(\boldsymbol{a} \cup \boldsymbol{b})^*\boldsymbol{ba}$ skapa en **NFA**. Den slutliga automaten byggs upp del för del och för varje delautomat visas ovanför respektive automat det reguljära uttryck mot vilket automaten svarar.



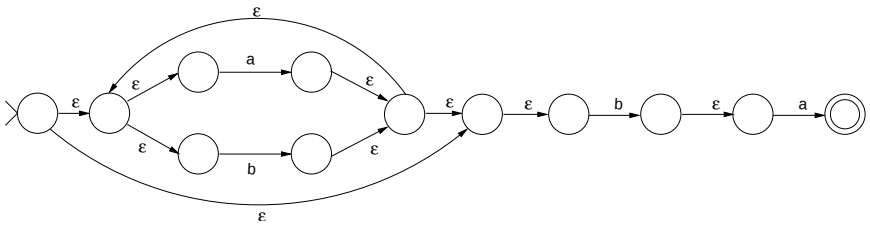
Kommentarer: De två första automaterna är de grundläggande automaterna som är väldigt enkla; de består båda av ett starttillstånd, ett accepterande tillstånd samt en transition mellan dessa som konsumerar ett a respektive ett b. Den tredje automaten är konkateneringen av dessa båda. I den skapas en ε-transition mellan sluttillståndet för automaten för b och starttillståndet för automaten för a, vilka därefter förlorar sina egenskaper. På så sätt erhålls alltså en konkatenering mellan de båda, alltså $\boldsymbol{ba}$.

$(\boldsymbol{a} \cup \boldsymbol{b})$ $(\boldsymbol{a} \cup \boldsymbol{b})^*$



Kommentarer: I den första automaten skapas en union mellan $\boldsymbol{a}$ och $\boldsymbol{b}$. Det görs genom att ett nytt start tillstånd och ett nytt sluttillstånd skapas. Därefter skapas ε-transitioner från det nya starttillståndet till starttillstånden för automaterna för $\boldsymbol{a}$ och $\boldsymbol{b}$ (vilka därefter förlorar sina egenskaper). Därefter skapas transitioner från de accepterande tillstånden för automaterna för $\boldsymbol{a}$ och $\boldsymbol{b}$ till det nya accepterande tillstånden samtidigt som de tidigare accepterande tillstånden förlorar sina egenskaper.
I den andra automaten skapas en automat för $(\boldsymbol{a} \cup \boldsymbol{b})^*$, vilket görs genom att på den första automaten skapa två ε-transitioner; ett från det accepterande tillståndet till sluttillståndet samt en åt andra hållet. Här har även ett nytt start- respektive sluttillstånd skapats.

$(\boldsymbol{a} \cup \boldsymbol{b})^* \boldsymbol{ba}$



Kommentarer: Den sista automaten är en sammanslagning av den tredje automaten av de översta tre och den andra automaten av de mellersta två. Här skapas den slutgiltiga automaten genom att en ε-transition mellan det accepterande tillståndet för automaten $(\boldsymbol{a} \cup \boldsymbol{b})^*$ och starttillståndet för automaten för $\boldsymbol{ba}$ skapas. I och med detta är algoritmen färdig och den slutgiltiga automaten erhålls.

4.2 Implementering av algoritmer

Att implementera de ovan angivna algoritmerna torde inte kräva speciellt mycket. Vad som behövs är dels en parser vilken kan parsa ett reguljärt uttryck utifrån given syntax men det är enkelt att göra med en parsergenerator. Därefter behövs funktioner för att skapa de olika automaterna. Dessa funktioner är enkla att göra; det enda som behöver göras är att lägga till alla ursprungliga tillstånd och transitioner och därefter skapa de nya tillstånd och transitioner som krävs enligt ovan och till sist sätta nya start- och sluttillstånd. Dessa är inte speciellt intressanta och någon närmare titt på dem behövs inte.

4.3 Parser

Några ord om parsern bör också nämnas. Parsern implementeras med hjälp av en parsergenerator. En sådan använder dels en mängd tokens, vilka i det här fallet är

- "(" och ")" för att skilja uttryck åt.
- "U" för union av två uttryck.
- "*" och "+" för Kleene star och plus-operatorn
- "LAMBDA", "EPSILON" och "E" (vilka betecknar ε)
- "a" ... "z".

samt en grammatik som ser ut enligt följande

$$
\begin{array}{lcl}
E & \rightarrow & U \cup E \mid U \\
U & \rightarrow & RU \mid R \\
R & \rightarrow & S^* \mid S^+ \\
S & \rightarrow & a \ldots z \mid (E)
\end{array}
$$

I och med grammatikens utseende kommer parsern implementeras på sådant sätt att hänsyn tas till operatorernas associativitet och inbördes prioritet vilket leder till korrekt parsade uttryck. Varje uttryck skapar en automat vilken därefter kan till exempel konverteras till **DFA** och minimeras för att kunna jämföras med andra automater.

5 Att jämföra två automater

Den kanske allra viktigaste funktionen i hela systemet är den som beskrivs härnäst; att jämföra en automat med en annan. Varför är det så viktigt? Det beror på att det är det här som hela förståelsen för begreppet ändliga automater ligger; att kunna förstå varför en automat accepterar samma språk som en annan, att kunna se likheten mellan en automat och ett reguljärt uttryck. Allt detta grundar sig på jämförelsen av två automater.

Hela syftet med ändliga automater överhuvudtaget är att beskriva språk och en mycket viktig funktion som AUTLAW har är att lära användaren hur denne skapar en automat som accepterar ett specifikt språk men även tvärtom, vilket språk en given automat accepterar. För att lära sig detta på ett bra sätt kan användaren exempelvis få i uppgift att skapa en ändlig automat som accepterar ett visst språk (som exempelvis beskrivs av en ändlig automat eller ett reguljärt uttryck). Systemet måste sedan kontrollera huruvida användarens svar är korrekt; systemet måste kunna verifiera att två automater accepterar samma språk (vilket är fullt tillräckligt då varje reguljärt uttryck ju kan omformas till en automat).

Som tur är kan varje automat göras om till en deterministisk ändlig automat som sedan kan minimeras[4]. Det är nämligen känt att två minimerade deterministiska automater accepterar samma språk om och endast om de är lika upp till tillståndens namn. Sådana automater kallas *isomorfa*.

Definition 5.1 (Isomorfa automater)

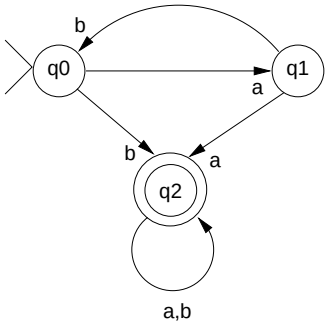
Två ändliga automater $M = (Q, \Sigma, \delta, F)$ och $M' = (Q', \Sigma, \delta', F')$ är isomorfa om det finns en bijektiv funktion $rename : Q \to Q'$ sådan att $F' = \{rename(q) | q \in F\}$ och för alla $q \in Q$ och $a \in \Sigma$ gäller $rename(\delta(q, a)) = \delta'(rename(q), a)$.

Två isomorfa automter accepterar samma språk.

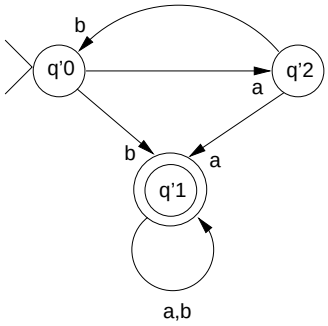
En förutsättning för att kunna jämföra två automater är att båda två är minimerade **DFA**er. Om så inte är fallet kan det hända att språken är desamma fastän autoamterna är olika.

[4] Se kapitel 3.4 på sid 27

Exempel



δ	a	b
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_2\}$



δ	a	b
q'_0	$\{q'_2\}$	$\{q'_1\}$
q'_1	$\{q'_1\}$	$\{q'_1\}$
q'_2	$\{q'_1\}$	$\{q'_0\}$

Som synes är automaterna inte identiska men det är enkelt att se att de är isomorfa och alltså accepterar samma språk. De ser identiska ut men det enda som skiljer är tillståndens ordning.

5.1 Implementation av jämförelse av automater

När det gäller jämförelse av automater är det viktiga inte att automaterna ser likadana ut utan att de accepterar samma språk. Vid en implementation av denna funktion är det viktigt att detta återspeglas; man kan inte enbart titta på tillstånden och transitionerna och säga att "de här automaterna är inte isomorfa för att transitionerna inte är lika".

Lösningen på problemet

Problemet är alltså hur man skall gå till väga för att ta reda på om två automater accepterar samma språk. Lösningen på problemet: para ihop identiska tillstånd genom att traversera automaterna.

Eftersom antalet tillstånd i de båda automaterna måste vara detsamma, är det inte helt omöjligt att i två identiska automater motsvarar ett tillstånd i en automat *alltid* ett enda tillstånd i den andra automaten. Om ett tillstånd i en automat skulle motsvara två eller fler tillstånd i den andra automaten kan man lista ut att automaterna inte är isomorfa och därför inte accepterar samma språk. Ett lämpligt sätt att implementera detta skulle vara att göra det rekursivt; om det inte är avgörbart att automaterna är isomorfa, använd transitionsfunktionerna för ett visst input-tecken och anropa funktionen igen fast nu med de ur transitionsfunktionen erhållna tillstånden.

En viktig sak att ha är någon typ av lista som håller reda på vilka tillstånd som svarar mot vilka. Varje gång ett nytt par skall läggas till kontrolleras huruvida tillståndet i den ena automaten svarar mot ett *annat* tillstånd i den andra automaten och om så är fallet är någonting galet på tok och funktionen kan returnera ett falskt värde. Listan bör självklart vara en central lista, eftersom

det intressanta är när listan innehåller lika många par av tillstånd som det finns tillstånd i automaten och alla input-tecken är undersökta; då är ju alla par av tillstånd identifierade och inga vägar finns kvar att undersöka. Då kan man sluta sig till att automaterna är identiska.

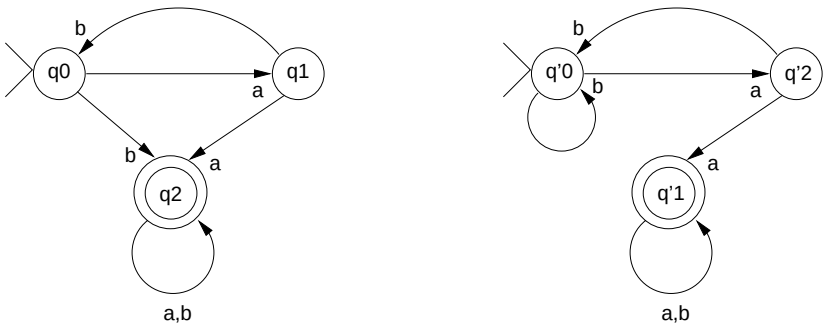
En rekursiv funktion som gör så skulle kunna se ut ungefär enligt följande pseudokod.

```
boolean compareStates(state1, state2, list_of_tuples)
{
  for (i = all input chars) {
    next1 = automaton1.get_transition(state1, input_char(i))
    next2 = automaton2.get_transition(state2, input_char(i))
    for (j = all tuples in list_of_tuples) {
      if one of next1 and next2 exists in a tuple and not the other
        return false
      else
      if (next1, next2) exists
        set tuple_found
    }
    if not tuple_found {
      list_of_tuples.add_tuple(next1, next2)
      if compareStates(next1, next2, list_of_pairs) == false
        return false
    }
  }
  return true
}
```

Exempel på algoritmens funktionalitet

Nu när en algoritm/funktion tagis fram kan det vara intressant att undersöka huruvida den fungerar eller ej. Detta görs med två automater som *ej* är identiska varvid funktionen bör returnera ett falskt värde.

Följande automater jämförs:



Att automaterna inte accepterar samma språk är ganska enkelt att se. Den första automaten accepterar till exempel strängen b vilket inte den andra automaten gör och man kan därför sluta sig till att automaterna inte är isomorfa.

Nu när det är fastslaget bör alltså även ovanstående algoritm komma fram till samma sak, det vill säga att automaterna inte är isomorfa. Något som inte nämnts är input till första funktionsanropet. Det är ganska självklart; automaternas starttillstånd. Ett anrop enligt ovanstående pseudokod skulle alltså vara `compareStates(init1, init2, list_of_tuples)` där `list_of_tuples` innehåller endast en tupel bestående av de två starttillstånden. I det här fallet blir denna tupel $\{q_0, q_0'\}$. Nedanstående visar hur algoritmen går till. Det som visas i tabellen är aktuella tillstånd, vilket input-tecken som används, vilka nästa tillstånd i första automaten respektive andra automaten är samt hur listan med tillstånd ser ut.

Steg	Tillstånd	Input	Nästa 1+2	Lista
1	q_0, q_0'	a	q_1, q_2'	$\{\{q_0, q_0'\}\}$
2	q_1, q_2'	a	q_2, q_1'	$\{\{q_0, q_0'\}, \{q_1, q_2'\}\}$
3	q_2, q_1'	a	q_2, q_1'	$\{\{q_0, q_0'\}, \{q_1, q_2'\}, \{q_2, q_1'\}\}$
4	q_2, q_1'	b	q_2, q_1'	$\{\{q_0, q_0'\}, \{q_1, q_2'\}, \{q_2, q_1'\}\}$
5	q_1, q_2'	b	q_0, q_0'	$\{\{q_0, q_0'\}, \{q_1, q_2'\}, \{q_2, q_1'\}\}$
6	q_0, q_0'	b	q_2, q_0'	$\{\{q_0, q_0'\}, \{q_1, q_2'\}, \{q_2, q_1'\}\}$

Kommentarer:

1. Paret $\{q_1, q_2'\}$ förekommer ännu inte i listan och kommer därför att läggas till listan över tupler. Antalet tupler är inte lika med antalet tillstånd varför `compareStates` anropas rekursivt med det nya paret.

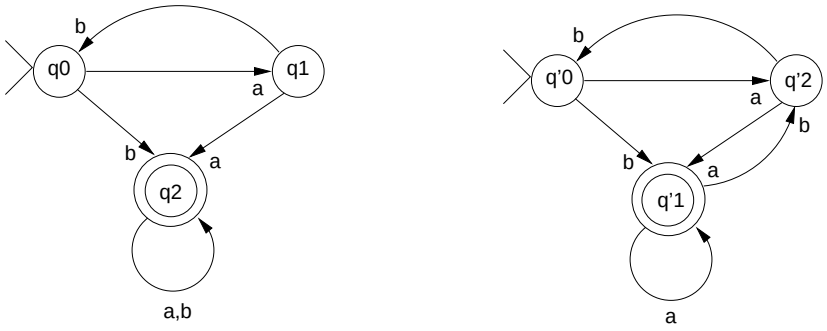
2. Inte heller paret $\{q_2, q_1'\}$ finns i listan och kommer därför att läggas till. Vid undersökning av b kommer `compareStates` att anropas för q_2, q_1'.

3-4. Nästa tillstånd är q_2, q_1' och här testas både a och b men båda dessa ger inga nya tupler att lägga till. Därför kommer `true` att returneras här.

5. Tillbaka till tillstånd q_1, q_2'. Nu testas istället b men det ger bara paret q_0, q_0' vilket existerar och därför kommer funktionen att returnera `true`.

6. Tillbaka till tillstånd q_0, q_0' och tecknet b. Nu erhålls paret $\{q_2, q_0'\}$ men eftersom q_2 redan finns i listan i par med q_1' betyder det att något har gått galet och därmed är automaterna inte isomorfa och funktionen returnerar `false`.

Det är mycket viktigt att algoritmen kontrollerar *alla* input-tecken och inte stannar så fort alla par har undersökts. Detta eftersom det mycket väl skulle kunna finnas transitioner som inte alls stämmer med varandra. Man får aldrig glömma att det inte bara är tillstånden som ska stämma överens utan även transitionerna. Följande två automater visar just hur viktigt det är att kontrollera *alla* möjliga vägar.



De två automaterna är mycket lika varandra men en enda transition skiljer dem åt. Om algoritmen hade nöjt sig med att stanna efter att ha parat ihop alla tillstånd i automaten skulle den förmodligen ha stannat efter bara två steg; då skulle nämligen paret $\{q_2, q_1'\}$ funnits som sista par och därmed skulle aldrig algoritmen ett par steg senare funnit paret $\{q_2, q_2'\}$ (då b undersöks från q_2) och därmed insett att de båda automaterna inte är isomorfa.

Mer information finns att ge

Om automaterna inte har samma antal tillstånd eller samma input-tecken är det enkelt att beskriva detta problem för användaren av systemet. Om däremot inte transitionsfunktionerna är identiska är det betydligt svårare att ge användaren en vettig förklaring hur automaterna skiljer sig åt. Vad som däremot skulle vara vettigt är att föreslå för användaren en sträng som skiljer de båda automaterna åt, det vill säga en sträng som accepteras av den ena automaten men inte av den andra. Användaren har därmed chans att följa en viss sträng genom automaterna och därmed själv se hur de båda automaterna följer sig åt. Algoritm för just detta presenteras härnäst.

5.2 Att skapa åtskiljande strängar

Hur ska man då gå till väga för att visa hur två automater skiljer sig åt? Ett sätt är att skapa slumpmässiga strängar och därefter testa dem i båda automaterna. Detta är dock mycket osäkert och framför allt inte en speciellt bra lösning. Man kan nu inse att det, utifrån de båda ingående automaterna, borde kunna gå att skapa en automat som accepterar antingen alla strängar som den första accepterar *eller* alla strängar som den andra accepterar men som inte accepterar strängar som *båda* accepterar. Frågan är bara: hur går man tillväga?

Det hela görs genom att skapa en automat där varje tillstånd är ett par av tillstånd; ett från vardera av de ursrpungliga automaterna, det vill säga en kartesisk produkt mellan de två tillståndsmängderna. Därefter skapas transitioner genom att för varje ingående ursprungligt tillstånd och för varje input-tecken skapa en transition till det par av tillstånd som svarar mot de transitioner som görs i de ursprungliga automaterna. Starttillstånd i den nya automaten är enkelt; det kommer att bli det tillstånd som utgörs av de två ursprungliga starttillstånden. Däremot är det svårare att bestämma vilka tillstånd som skall vara sluttillstånd.

Det hela avgörs beroende på vilket syfte automaten skall användas. I detta fall skall en sträng bara accepteras av den ena ursprungliga automaten men inte den andra. Sluttillstånden blir då den mängd av tillstånd som svarar mot de par av tillstånd där ena tillståndet är ett sluttillstånd i den ena automaten men det andra är ett icke-accepterande tillstånd i den andra automaten. Rent formellt kan denna automat beskrivas som enligt nedan.

Definition 5.2 (Produktautomat)

Antag att $M = \{Q, \Sigma, \delta, q0, F\}$ och $M' = \{Q', \Sigma, \delta' q0', F'\}$ är två minimerade deterministiska automater som betecknar språken $L(M)$ och $L(M')$. Då definieras automaten N som accepterar alla strängar som ingår i antingen $L(M)$ eller $L(M')$ men inte både och enligt följande:

$$M'' = \{Q'', \Sigma, \delta'', q_0'', F''\}$$

där

$$\begin{aligned} &Q'' = Q \times Q' \\ &\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a)) \ \forall \ a \in \Sigma \ och \ (q, q') \in Q'' \\ &F'' = \{(f, f') | (f \in F \wedge f' \notin F') \vee (f \notin F \wedge f' \in F')\} \end{aligned}$$

Det är uppenbart att antalet tillstånd som den nya automaten N ovan får är $card(Q) \cdot card(Q')$, som kommer att bli en jämn kvadrat då Q och Q' har lika många tillstånd, och att från varje tillstånd kommer exakt en transition för vardera input-tecken att gå.

En algoritm för framtagandet av en dylik automat är inte speciellt avancerad; den skapar nya tillstånd genom att para ihop varje tillstånd i den ena automaten med tillstånden i den andra. Därefter skapas transitioner genom att slå ihop tillståndet som erhålls vid transition i den ena automaten med tillståndet som erhålls vid transition i den andra automaten. Till sist sätts sluttillstånd genom att ta alla par av tillstånd där ena tillståndet är sluttillstånd i den ena automaten medan det andra *inte* är sluttillstånd i den andra automaten. Input-tecken för automaten är givetvis desamma.

Exempel

Antag att de icke-isomorfa automaterna enligt ovan skall jämföras och en automat som accepterar strängar som accepteras av den ena automaten men inte av den andra skall skapas. Dessa automater har följande transitionsfunktioner:

Q

δ	a	b
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_2\}$

Q'

δ	a	b
q_0'	$\{q_2'\}$	$\{q_0'\}$
q_1'	$\{q_1'\}$	$\{q_1'\}$
q_2'	$\{q_1'\}$	$\{q_0'\}$

Ska nu en ny automat M'' skapas enligt algoritmen ovan erhålls:

$$Q'' = Q \times Q' = \{q_0q_0', q_0q_1', q_0q_2', q_1q_0', q_1q_1', q_1q_2', q_2q_0', q_2q_1', q_2q_2'\}$$

och transitionsfunktionen kommer att se ut enligt följande:

δ	a	b
q_0q_0'	q_2q_1'	q_0q_2'
q_0q_1'	q_2q_2'	q_0q_0'
q_0q_2'	q_2q_2'	q_0q_2'
q_1q_0'	q_1q_1'	q_1q_2'
q_1q_1'	q_1q_2'	q_1q_0'

δ	a	b
q_1q_2'	q_1q_2'	q_1q_2'
q_2q_0'	q_1q_1'	q_0q_2'
q_2q_1'	q_1q_2'	q_0q_0'
q_2q_2'	q_1q_2'	q_0q_2'

Vilka av dessa är då accepterande tillstånd? Jo de tillstånd som innehåller ett accepterande tillstånd i Q och ett icke-accepterande tillstånd i Q' eller vice versa. Inspektion av detta ger att

$$F'' = \{q_0q_2', q_1q_0', q_1q_1', q_2q_2'\}$$

OK, då skall automaten vara klar. Det måste betyda att den erhållna automaten accepterar strängar som accpeteras av den ena av Q och Q' men inte av den andra. Det kan ju enkelt testas. En sträng som accepteras av automaten ovan är exempelvis *abba*. Det måste betyda att den ena av Q och Q' ej accepterar denna sträng. Ett test vore kanske på sin plats?

Beräkningarna *abba* för de båda automaterna ser ut enligt följande:

	$[q_0, abba]$		$[q_0', abba]$		$[q_0q_0', abba]$
$\vdash$	$[q_1, bba]$	$\vdash$	$[q_2', bba]$	$\vdash$	$[q_2q_1', bba]$
$\vdash$	$[q_0, ba]$	$\vdash$	$[q_0', ba]$	$\vdash$	$[q_0q_0', ba]$
$\vdash$	$[q_2, a]$	$\vdash$	$[q_0', a]$	$\vdash$	$][q_0q_2', a]$
$\vdash$	$[q_2,\varepsilon]$	$\vdash$	$[q_1',\varepsilon]$	$\vdash$	$[q_2q_2', \varepsilon]$
	accepteras		förkastas		accepteras

Som väntat har den ena automaten accepterat strängen *abba* medan den andra har förkastat den. Det innebär att automaten fungerar och man kan, lite våghalsat, dra slutsatsen att algoritmen fungerar.

Att ta fram strängar

Nu när en automat som accepterar strängar som den ena av de två ursprungliga automaterna accepterar har tagis fram återstår endast att ta fram en mängd strängar som kan användas som exempel för användaren. Detta torde inte vara speciellt svårt, eller? Svaret på frågan är: det beror på hur långa strängar som skall tas fram. I det allra enklaste fallet görs en djupet-först-sökning som bygger

upp en sträng och så fort ett accepterande tillstånd funnits, avslutas sökningen och strängen returneras. I en sådan sökning accepteras inga loopar, det vill säga att ett besökt tillstånd får inte besökas igen.

Om man så önskar, vilket har skett vid implementeringen av AUTLAW, kan ett lite mer avancerat tillvägagångssätt användas, nämligen att ett besökt tillstånd får besökas igen. Detta kan varieras till att exempelvis innefatta att ett slumpmässigt antal besök vid samma tillstånd tillåts och att en räknare associeras med varje tillstånd som håller reda på hur många gånger tillståndet besökts. Fortfarande returneras strängen så fort ett accepterande tillstånd hittats. Detta tillvägagångssätt ger något längre strängar.

För att göra det riktigt avancerat kan en algoritm implementeras som tillåter accepterande tillstånd att besökas flera gånger. Det gör att strängarna kan bli långa; i värsta fall kanske för långa vilket kanske kan göra dem lite för jobbiga att följa för en användare som inte är speciellt van vid ändliga automater. Av denna anledning, och även på grund av att det kräver betydligt mer tid än vad som är nödvändigt, har inte detta tillvägagångssätt används vid implementation av AUTLAW; det räcker fullt och gott med att tillåta upp till tre besök vid av samma tillstånd vid skapandet av exempel på strängar som accepteras av den ena automaten men inte den andra.

6 Att spåra en sträng

Att spåra en sträng, "spåra" i bemärkelsen att följa strängens väg genom tillståndsdiagrammet och för varje tecken som konsumeras byta tillstånd, är den funktion som är kanske mest "synlig" utåt och som användaren har enklast att relatera till. Detta i och med att användaren själv får följa med och får steg för steg inspektera hur automaten byter aktuellt tillstånd då ett visst input-tecken konsumeras. De andra funktionerna gör mycket i det tysta och har en massa hyss för sig i bakgrunden som användaren inte ser och som kanske inte är så lätt att förstå; ta exempelvis konvertering från ett reguljärt uttryck till en **NFA**-ε som ju på något mystiskt sätt konverterar unioner, konkateneringar och kleene star-operationer till en mängd tillstånd med en massa transitioner och har man inte riktigt förstått teorin bakom kanske man inte blir mycket klokare.

Att spåra en sträng, däremot, är nog betydligt enklare och mer jordnära för en användare. Här får man chansen att verkligen se strängen konsumeras och följa steg för steg och hela tiden veta vilket som är aktuellt tillstånd.

6.1 Den enklaste av implementering

Att följa en strängs vandring genom en automat är förmodligen den allra enklaste av funktioner att implementera och kräver inte speciellt mycket till algoritm; detta förstås beroende på hur avancerat man vill göra det. Som det kommer att visa sig kommer AUTLAW att inte bara visa aktuellt tillstånd utan även visa vilka ε-transitioner som kan göras.

Den allra enklaste implementeringen av funktionen är dock att ta ett input-tecken i taget och visa vilket tillstånd som är aktuellt tillstånd genom att exempelvis visa det med en annan färg. Skulle automaten vara en **DFA** kommer funktionen att bli ytterst enkel; detta eftersom varje sträng endast kan ta *en enda* bestämd väg genom tillståndsdiagrammet för en **DFA**. Är automaten en **NFA** blir det lite svårare, men inte så mycket, eftersom programmet då måste hålla reda på en mängd av tillstånd och inte bara ett enda.

Att spåra en sträng genom en DFA

Det allra enklaste är som tidigare nämnts att spåra en sträng genom en **DFA**. Det beror på att varje godtycklig sträng, oavsett om den finns med i språket som automaten representerar eller ej, alltid kommer att ta en speciellt väg genom grafen; detta i och med att automaten är deterministisk och för varje tillstånd och input-tecken finns endast en väg att gå. Det är sedan väldigt enkelt att avgöra huruvida strängen accepteras eller ej; om det sista tillståndet är ett accepterande tillstånd kommer strängen att accepteras, annars ej.

En funktion som spårar en sträng genom en **DFA** skulle kunna se ut enligt nedanstående pseudokod.

```
int [] trace_string(a_string)
{
  int [a_string.length()+1] a_list_of_states

  current_state = automaton.get_initial()
  a_list_of_states[0] = current_state
  for (i = each char in a_string) {
    a_list_of_states[i] = current_state;
    current_state = automaton.get_transition(current_state, i)
  }
  return a_list_of_states
}
```

Som synes är algoritmen inte speciellt lång och svår och den är väldigt rättfram. Den tar helt enkelt reda på vilken väg genom grafen som den aktuella strängen kan ta och lagrar dessa värden i en lista. Detta för att på ett enkelt och smidigt sätt låta användaren gå framåt och bakåt i diagrammet.

6.2 Mer avancerad implementering

Den ovan angivna algoritmen är enklast möjlig och den fungerar på alla **DFA**er eftersom dessa, som namnet anger, är deterministiska och det går alltid att avgöra vilken väg strängen tar genom diagrammet. Nästa steg är att göra det hela lite mer komplicerat, nämligen att göra samma sak i såväl **NFA**er som **NFA**-ε, av vilka den senare kräver lite mer omtanke än den första.

Att spåra en sträng genom en NFA

Att spåra en sträng genom en **NFA** är egentligen inte *så* mycket svårare än att spåra en sträng genom en **DFA**. Skillnaden är bara att tillstånden inte samlas i en lista utan i en lista av listor; detta eftersom varje tillstånd kan ge fler tillstånd att gå till för varje input-tecken och inte bara ett som för en **DFA**. När det gäller **NFA** är det också viktigt att komma ihåg att en sträng inte accepteras då *samtliga* vägar undersökts och inte någon av dem slutar i ett accepterande tillstånd. Dessutom är det bra att veta att om det inte finns en väg från ett tillstånd för ett visst input-tecken så avbryts beräkningen där och den "grenen" blir så att säga ogiltig; endast de beräkningar som konsumerar hela strängen och stannar i ett accepterande tillstånd är giltiga.

Nedan visas pseudokod på hur en funktion som spårar en sträng genom en **NFA** kan se ut.

```
list [] trace_string(a_string)
{
  list [a_string.length()+1] a_list_of_lists
  list[0] = new list();

  list[0].add(automaton.get_initial());
  for (i = each char in a_string) {
    list[i+1] = new list();
    for (j = all states in list[i])
      list[i+1].add(automaton.get_transition(j, i));
  }
  return a_list_of_lists
}
```

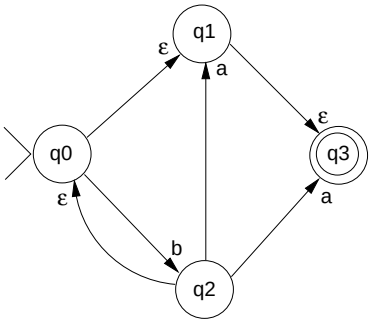
Funktionen returnerar en lista som innehåller listor på vilka tillstånd som är aktuella för vart och ett av de input-tecken som ingår i strängen. Även här görs detta för att underlätta stegning framåt och bakåt i grafen.

Att spåra en sträng genom en NFA-ε

Den mest allmänna funktionen att implementera är att spåra en sträng genom en **NFA**-ε, vilket beror på att då måste hänsyn tas inte bara till alla de vanliga transitioner som kan göras utan även till ε-transitioner. Det smidigaste sättet att göra det på är att först finna ε-closure för vart och ett av de tillstånd som ingår i den mängd som består av aktuella tillstånd och därefter inkludera dem; antingen genom att visa direkt att det går att gå till och därmed göra dem till en del av mängden med aktuella tillstånd, eller genom att göra en egen mängd av dessa tillstånd och därmed särskilja dem från de "ursprungliga" tillstånden som erhålls.

Väljs det första sättet som beskrivs ovan särskiljs inte tillstånden som nås "direkt" och de som nås via ε-transitioner; alla dessa slås ihop till en enda mängd av nåbara tillstånd. Om å andra sidan det andra sättet väljs krävs en extra lista, en "ε-closure-lista" som innehåller ε-closure för alla de tillstånd som ingår i den ursprungliga listan.

Fördelen med den senare metoden är att det går att grafiskt särskilja de tillstånd som nås direkt och de som nås med ε-transitioner, exempelvis genom att färga dem i olika färger. I och med det kan dock problem uppstå. Beakta exempelvis följande automat:



Antag att en sträng som börjar med *ba* ska spåras (hur den fortsätter är ovidkommande för tillfället). Efter att första tecknet konsumerats befinner sig spårningen i tillstånd q_2; detta eftersom det är den enda transitionen som konsumerar ett b. När sedan a konsumeras kommer spårningen att hamna i antingen q_1 eller q_3 varför dessa färgas på ett speciellt sätt. Här dyker dock ett problem upp: q_3 finns ju även med i ε-closure för q_1 och skall därmed färgas med en annan färg. Problemet är: vilken färg skall tillståndet ha?

Eftersom AUTLAW syftar till att vara så enkelt och tydligt som möjligt finns två möjligheter; antingen kombineras färgerna på något sätt så att det framgår att q_3 kan nås direkt eller via q_1 och sedan en ε-transition eller så prioriteras ett av visningssätten. Det senare är det mest logiska att göra och därför kommer direkta transitioner att prioriteras före ε-transitioner; det känns mer intressant att veta vilka tillstånd man kommer till direkt än vilka man kan komma till genom ε-transitioner. Om man kommer till ett tillstånd på båda sätten känns det mer intressant med den direkta varianten.

Nu återstår bara en enda sak; att försöka ta fram en vettig algoritm som enkelt tar reda på vilka vägar en viss sträng tar genom en **NFA**-ε. Den algoritm som användes för **NFA** utan ε-transitioner kan enkelt modifieras till att även inkludera ε-closure för varje tillstånd. Pseudokod för detta kan se ut enligt nedan.

```
class list_object {
  list a_list_of_states
  list a_ep_closure
  ..
  /* and some methods to go with that */
}

list_object [] trace_string(a_string)
/* Automaton found in variable a */
{
  list_object[a_string.length()] a_list_object

  a_list_object[0].add_state(a.get_initial());
  a_list_object[0].find_closure();
  for (i = each char in a_string) {
    a_list_object[i+1] = new list_object();
```

```
    for (j = all states in a_list_object[i].a_list_of_states)
      a_list_object[i+1].add_state(a.get_transition(j, i));
    for (j = all states in a_list_object[i].a_ep_closure)
      a_list_object[i+1].add_state(a.get_transition(j, k));
    a_list_object[i+1].find_closure();
  }
  return a_list_object
}
```

Funktionen ovan kommer att ge två olika listor; en innehåller de tillstånd vilka erhålls direkt vid en transition medan den andra listan innehåller ε-closure för tillstånden i den första listan. För att visualisera det hela kan nu exempelvis alla tillstånd i den ena listan visas med en färg och tillstånden i den andra med en annan färg. Användaren kan då enkelt se steg för steg både vilka tillstånd som nås direkt vid en transition och vilka tillstånd som kan nås via en ε-transitioner. När sista bokstaven konsumerats kan direkt avgöras huruvida strängen accepteras eller ej; detta görs om det finns något accepterande tillstånd i någon av de två listorna.

Sammanfattning

I och med den sista av de tre algoritmerna för att spåra en sträng har samtliga funktioner som från början var tänkt att undersöka undersökts och algoritmer för dessa har tagits fram. Som tidigare nämnts har algoritmer för att konvertera reguljära grammatiker till automater och vice versa utelämnts helt och hållet. Vidare har även algoritmer för konvertering från automater till reguljära uttryck utelämnats eftersom det rent pedagogiskt inte är så viktigt inom ramen för systemet. Dessutom skulle uttrycken kunna bli stora och oöverskådliga.

Vad har då tagits fram? Svaret är de mest grundläggande ingredienserna för att skapa ett system som har som huvudsaklig uppgift att öka förståelsen för och användandet av formella språk i allmänhet och ändliga automater i synnerhet. Detta inkluder aralgoritmer för att konvertera mellan olika typer av automater och minimera dem, algoritmer för att parsa reguljära uttryck samt konvertera dessa till automater, algoritmer för att jämföra automater med varandra och ta fram strängar som skiljer icke-isomorfa automater åt samt slutligen algoritmer för att testa huruvida en sträng accepteras av en automat och hur detta kan visas detta grafiskt.

Nu återstår bara att implementera systemet och dess algoritmer.

7 AUTLAW - systemet

Att skapa ett system av AUTLAWs magnitud kräver hel del planering. En stor del av denna har redovisats i denna rapport; bland annat har de algoritmer som används av programmets funktioner detaljerat presenterats och exempel på hur de använts har visats. Att implementera systemet kräver också en hel del och det återstår många frågor att besvara; vilket programspråk skall användas? Vad skall systemet bestå av? Hur skall formatet var? Detta kapitel är tänkt att besvara några av de frågor som rör implementeringen samt presentera det grundläggande systemet, innan ett GUI skapas.

7.1 Implementationsspråk

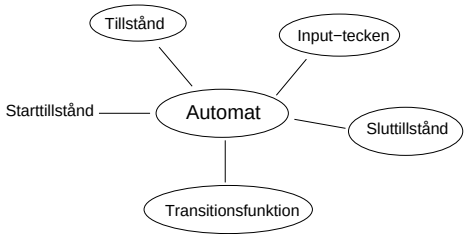
En viktig fråga att besvara innan implementation påbörjas är vilket programspråk som skall användas. Det finns ju en uppsjö att välja ifrån men eftersom den föredragna utvecklingsmiljön är Linux/Unix men programmet lika gärna skall kunna användas under andra plattformer, exempelvis Windows, återstår egentligen bara ett: Java.

Med Java finns möjligheten att implementera ett system som är oberoende av utvecklingsplattform. För Java finns dessutom en överlägsen tutorial i form av java.sun.com; en website som utelämnar i princip ingenting. Eftersom kunskaperna i Java innan implementationens påbörjan var i princip minimala var en bra tutorial ett mer eller mindre ett krav, ett krav som Java gott och väl uppfyller, inte minst vad gäller GUI-programmering.

Ett annat skäl till att använda Java är att systemet lämpar sig väl att implementera objektorienterat, vilket gör java till den självklara kandidaten.

7.2 Automaten - Det mest centrala

Det mest centrala objektet i systemet är självklart automaten; oavsett vilken typ den har består varje automat av fem grundläggande delar: en mängd tillstånd, en mängd input-tecken, en transitionsfunktion, ett starttillstånd samt en mängd sluttillstånd. Eftersom en **DFA** är en speciell typ av **NFA** behöver systemet inte skilja mellan dessa två utan dessa egenskaper kan avgöras direkt utifrån automatens transitionsfunktion.



Alla de grundläggande delarna, förutom starttillstånd, är implementerade som generella länkade listor för göra dem så dynamiska som möjligt. Till varje lista finns en mängd metoder som opererar på varje objekts data; det finns metoder för att räkna data, hämta data, lagra data, ta bort data, skriva ut data etc. Dessa gör att listorna är mycket enkla och används till långt fler detaljer än till de ovan beskrivna, exempelvis för closure-funktioner och till spårfunktioner.

Transitionsfunktionen

Den mest svårimplementerade detaljen i systemet är transitionsfunktionen; den är dessutom den kanske viktigaste. Funktionen är en länkad lista i tre dimensioner; detta beror på att varje tillstånd har en transitionsfunktion för varje input-tecken (det gör de två första tillstånden) och dessutom kan varje tecken ha flera tillstånd - kom ihåg att automaterna är **NFA**er och därmed kan transitioner ske till flera olika tillstånd för varje input-tecken. Detta ger tre dimensioner.

Som tur är finns en mängd bra metoder för att operera på transitionsfunktionerna implementerade. Dessa gör det avsevärt lättare att använda funktionerna, oavsett vilket syfte användningen har.

Objektorienterad programmering

Objektorienteringens fördelar gör sig klart påminda vid implementeringen av AUTLAW. Varje funktion är en egen klass och systemets utformning gör det enkelt att finna generella klasser som sedan kan göras mer specificerade. Eftersom de ingående delarna i sig är implementerade klasser sker i princip ingen hantering av dessa direkt under automaten utan sker i respektive objekt. De funktioner automaten istället innehåller är de som opererar direkt på automaten; exempelvis konverteringsfunktioner, testfunktioner och jämförelsefunktioner. Dessutom innehåller automaten de operatorer reguljära uttryck tillhandahåller.

Totalt innehåller systemet omkring 50 olika klasser.

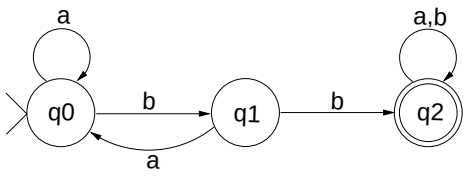
7.3 Parsergenerator

I implementeringen används en parsergenerator vid två olika tillfällen; dels för att parsa en automat och dels för att parsa reguljära uttryck. Varje automat sparas som ren text (se mer om automatens format under kapitel 7.4 nedan) och för att översätta denna till en automat, används en parsergenerator vid namn *javacc*. För att kunna skapa automater från reguljära uttryck behövs dessutom en parser för denna funktion. Generatorn arbetar såsom en parsergenerator bör; den behöver en mängd tokens och en grammatik att arbeta utifrån. I generatorn finns sedan möjlighet att lägga in egen kod där det behövs. Se kapitel 4.3 på sid 37 för mer information.

7.4 Automatens format

För att kunna spara automater i systemet i avseende att fortsätta arbeta med dem vid senare tillfälle, behövs ett format för detta. Det finns flera olika typer av format men det allra enklaste är förmodligen att använda ren ASCII-text; detta eftersom det då finns möjlighet att skapa automater inte bara i systemet utan även med hjälp av en texteditor.

Automaten har, som tidigare nämnts, fem olika delar vilka alla måste sparas på något sätt. Listorna med tillstånd, input-tecken och sluttillstånd samt start-tillståndet är inga problem att spara. Transitionsfunktionen är egentligen inte heller något problem; det gäller bara att komma ihåg dess tredimensionalitet. Transitionsfunktionen är alltså en lista med listor av listor med tillstånd. En enkel automat som exempelvis



skulle enligt formatet ovan, sparas i text enligt följande:

```
automaton("New automaton")
{ "q0", "q1", "q2" },
{ 'a', 'b' },
{
   { {0}, {1}, {} },
   { {0}, {2}, {} },
   { {2}, {2}, {} }
},
{ 0 },
{ 2 }
```

Tillstånden och input-tecknen lagras som strängar. Transitionsfunktionen samt start- och sluttillstånd lagras som nummer som motsvarar tillstånden i tillståndslistan. Det är här enkelt att se att automaten är en **DFA** eftersom för varje tillstånd och för varje tecken finns exakt ett tillstånd i funktionen. Den sista mängden på varje rad i transitionsfunktionen motsvarar transition för ε och eftersom automaten är en **DFA** finns inte några ε-transitioner.

Utöver de fem delarna finns även en beskrivning som är en sträng som är tänkt att beskriva vilken funktion automaten har, vilket språk den accepterar och så vidare. Dessutom finns möjlighet att spara undan positionen på vilken tillståndet placerats i GUIet. Tillståndens position läggs till sist i automaten.

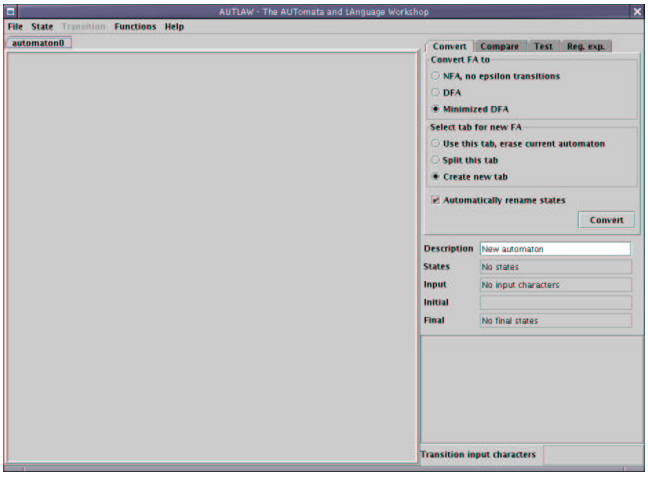
8 AUTLAW - GUI

Eftersom systemt är tänkt att användas i undervisningssyfte med betoning på ändliga automater är det självklart viktigt att systemet har ett bra GUI som är intuitivt och praktiskt. Funktioner skall vara lätta att hitta och de skall vara lätta att använda, tillstånd och transitioner skall vara enkla att lägga till, ta bort och redigera och information till användaren skall vara tydlig och enkel.

För att GUIet skall vara så tydligt som möjligt så åtskiljs själva automaten och dess "rityta" från funktionerna men dessa finns ändå nära till hands. Detta åstadkoms genom att dela in huvudfönstret i tre olika delar:

- Ritarean, vari automaterna ritas och redigeras.
- Funktionsarean, vari funktionerna återfinns.
- Informationsarean, vari information om automaten återfinns.

Såväl ritarean som funktionsarean består av flikar för att på så sätt gömma icke använda automater och funktioner. Vid start av programmet ser huvudfönstret ut enligt följande bild.



Längst upp i fönstret återfinns huvudmenyn. Denna förklaras senare.

8.1 Ritarean

Den största ytan av fönstret tas upp utav ritarean. På ritarean har användaren möjlighet att skapa och redigera sina automater genom att:

- Lägga till tillstånd och transitioner
- Radera tillstånd och transitioner

- Byta namn på tillstånd
- Ändra input-tecken på transitioner samt lägga till/ta bort ε-transitioner.
- Göra tillstånd till start- eller sluttillstånd eller ta bort dessa egenskaper.

Tillstånd flyttas genom att klicka och hålla nere musknappen och därefter dra tillståndet till den plats som önskas.

Markera flera tillstånd

På ritytan har användaren även möjlighet att markera flera tillstånd samtidigt; detta kan vara bra om man önskar flytta flera samtidigt utan att förändra deras inbördes positioner. Detta kan göras på ett flertal sätt, till exempel genom att klicka och hålla ner vänster musknapp för att därefter dra ut en rektangel som täcker de önskade tillstånden eller genom att markera tillstånd efter tillstånd genom att hålla ner <Ctrl>-knappen och samtidigt klicka på de önskade tillstånden.

Justera positioner

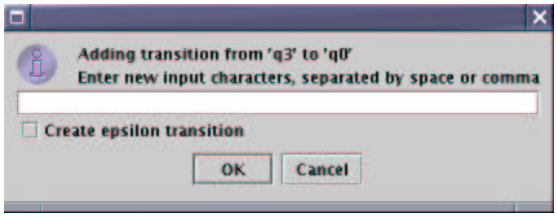
Det finns möjlighet för användaren att justera tillståndens inbördes positioner jämtemot varandra. Detta görs genom att markera de önskade tillstånden (se ovan) och därefter välja `State` | `Align` och i den meny välja det som passar. Det går att justera tillstånden i vertikal-led och horisontell-led och tillstånden justeras alltid efter den högsta/lägsta eller den längst till vänster/längst till höger.

Popup-meny

Vid högerklick dyker en popup-meny upp. Den menyn innehåller genvägar till de allra vanligaste kommandona; lägga till, ta bort och ändra tillstånd och transitioner samt att spara automaten. Den är mycket praktisk att använda och gör det enkelt att snabbt skapa de önskade automaterna.

Skapa transitioner

Att skapa transitioner mellan två tillstånd kan gå till på två sätt; den ena är att först markera ett från-tillstånd och därefter högerklicka och välja `Add transition` eller att välja `Transition` | `Add transition` från huvudmenyn. Nu dyker en pil upp och det är bara att klicka på det tillstånd som är till-tillstånd. När detta gjorts visas följande dialogruta.



I textfältet anges de input-tecken som transitionen gäller, åtskilda av komma eller mellanslag. Om en ε-transition önskas markeras kryssrutan märkt "Create epsilon transition". Klicka på **OK** och transitionen skapas.

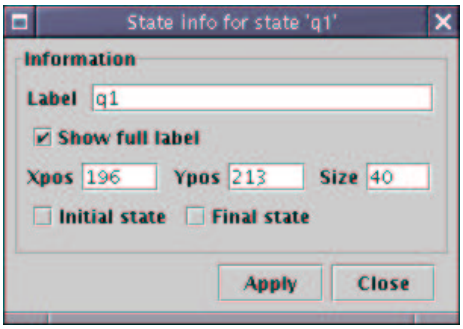
Det andra sättet att skapa en transition är att markera två tillstånd och välja `Transition` | `Add transition` från menyn och därefter göra enligt ovan. Nackdelen med detta tillvägagångssätt är att den transition som skapas *alltid* går från det lägst numrerade tillståndet till det högst numrerade tillståndet. Om en transition i omvänd ordning önskas måste den skapas enligt ovanstående metod.

Ändra transitioner

Om input-tecknen till en transition behöver ändras, markeras den transition som skall ändras och därefter väljs antingen `Transition` | `Edit input characters` från huvudmenyn eller `Edit input characters` från popup-menyn. En dialogrutan liknande den som visas ovan dyker upp och nu finns möjligheten att skriva in de input-tecken som önskas samt lägga till/ta bort ε-transition. Det går även bra att dubbelklicka på en transition för att ändra dess input-tecken.

Ändra tillstånd

Det finns flera sätt att ändra ett tillstånd i menyn. Vid dubbelklick på ett tillstånd visas följande dialogruta.



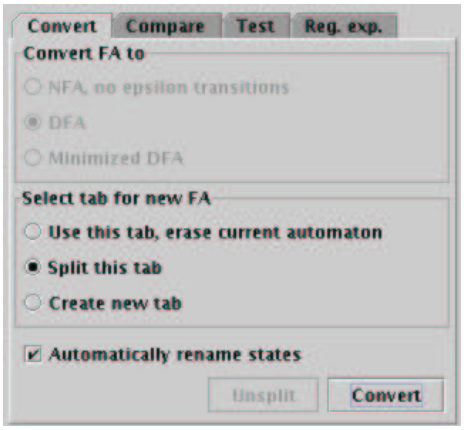
I den finns möjlighet att ändra de flesta av tillståndets egenskaper såsom dess position och storlek, namn samt om det skall vara start- eller sluttillstånd. Ändringar verkställs vid klick på **Apply** och fönstret stängs med klick på **Close**.

8.2 Funktionsarean

Systemet har fyra huvudsakliga funktioner uppdelade på fyra olika flikar. Var och en av funktionerna motsvarar de ovan beskrivna; konvertering mellan olika typer, konvertering av reguljärt uttryck till automat, jämförelse mellan automater samt test av en sträng.

Konvertering av automat

Fönstret för att konvertera en automat till en annan ser ut enligt nedan.



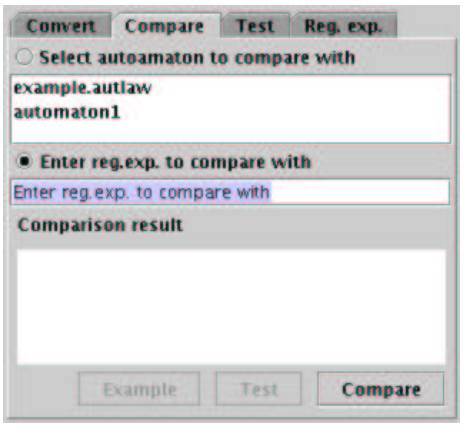
Funktionen är relativt enkel att använda. Användaren väljer först vilken typ av automat resultatet skall bli. Man bör observera att det inte går att konvertera "baklänges", det vill säga en minimerad **DFA** kan inte konverteras till en **NFA**; detta eftersom en **DFA** redan är en **NFA**, fast bara mer specialicerad.

Därefter väljer användaren vilken flik som den nya automaten kan hamna i. Det kan vara antingen i samma flik, varpå existerande automat kommer att raderas, i en ny flik eller så kan aktuellt fönster delas upp i två delar där originalet visas överst och den nya automaten underst. Om fönstret delas stryps vissa funktioner och dessa blir inte tillgängliga igen förrän användaren "odelat" (*unsplit*) fönstret; den understa automaten tas då bort och den övre, huvudautomaten blir ensam i fönstret igen.

När en automat konverteras kan användaren välja att automatiskt ge nya namn till den nya automatens tillstånd. Om detta inte väljs kommer den nya automatens tillstånd att få namn som speglar hur den är uppbyggd; man måste komma ihåg att en **DFA** byggs upp genom att bilda grupper av tillstånd från en **NFA**.

Jämförelse av automater

Fönstret för att jämföra två automater eller automat med reguljärt uttryck ser ut enligt nedan.

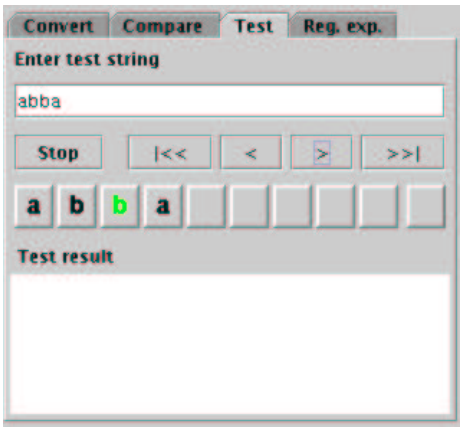


Att jämföra två automater är kanske den viktigaste funktionen av dem alla. Denna funktion låter användaren undersöka om den automat som skapats är likadan som en annan, kanske förutbestämd, automat.

Användaren väljer om automaten som skapats skall jämföras med en annan automat eller mot ett reguljärt uttryck. När sedan automaterna jämförts presenteras information om resultatet. Är de olika anges hur de skiljer sig åt och dessutom ges möjlighet för användaren att få exempel på strängar som skiljer automaterna åt, det vill säga som accepteras av den ena men inte den andra. Dessa strängar kan även testas under testfunktionen.

Test av sträng

Fönstret för att testa en sträng ser ut enligt nedan.



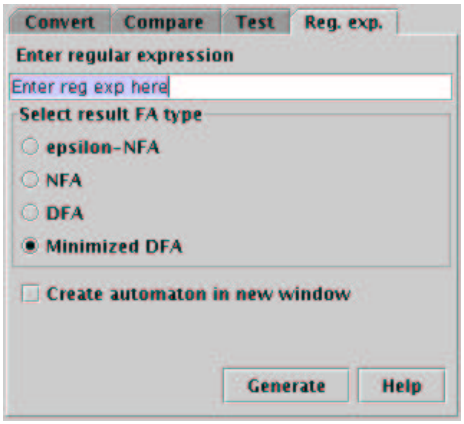
Funktionen för att följa en sträng genom en automat är också väldigt viktig och ger användaren möjlighet att se hur en sträng testas steg för steg, tecken för tecken. När hela strängen konsumerats ges information huruvida strängen accepterats eller ej. Det är möjligt att stega såväl framåt som bakåt i strängen

och det finns även möjlighet att testa hela strängen på en gång och direkt få resultatet.

Själva spårningen påbörjas genom att klicka **Init**. Då byter programmet läge till "spår-läge" och det är i detta läge inte möjligt att ändra i automaten. Däremot går det självklart att byta flik till en annan automat. Detta läge bibehålls till användaren klickar på **Stop**.

Skapa automat utifrån reguljärt uttryck

Fönstret för att skapa automat utifrån ett reguljärt uttryck ser ut enligt nedan.



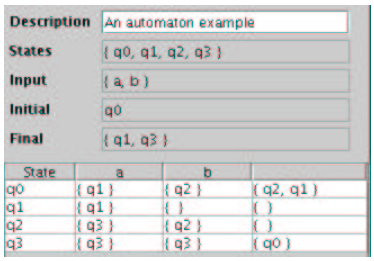
Den sista funktionen är den funktion som skapar en automat utifrån ett reguljärt uttryck. Liksom vid konvertering ges användaren här möjlighet att bestämma vilken typ den resulterande automaten skall ha samt huruvida automaten skall skapas i aktuell flik, och därmed radera existerande automat, eller om den skall skapas i ett nytt fönster.

Den observante ser att denna funktion finns på två ställen; skillnaden är att i denna funktion skapas en automat som visas i ritarean vilket inte är fallet då en automat jämförs med ett reguljärt uttryck. För att kunna inspektera skillnader mellan en automat och ett reguljärt uttryck är det således nödvändigt att använda denna funktion för att skapa en automat utifrån uttrycket och därefter jämföra automaterna med varandra. Är man bara intreserad av att veta om automaten motsvarar ett uttryck kan lika gärna funktionen under flik för jämförelse användas.

8.3 Informationsarea

Den sista arean som huvudfönstret består av är informationsarean. Här visas information om den aktuella automaten, den automat som visas i aktuell flik. Här har användaren även möjlighet att skriva in en beskrivning av automaten om så önskas. Informationen som visas är vilka tillstånd och input-tecken som finns,

vilka tillstånd som är start- respektive sluttillstånd samt slutligen information om transitionsfunktionen.



Informationen om transitionsfunktionen är interaktiv. Här finns nämligen möjligheten att markera ett eller flera tillstånd. När ett tillstånd i listan markeras, markeras även tillståndet i grafen och vice versa. Det gör att informationen blir mer läsbar och man har bättre chans att se vilka transitioner ett visst tillstånd har.

Längst ner visas information om vilka input-tecken markerad transition har; det kan på grund av GUIets utseende ibland vara svårt att se vilka input-tecken en transition har och denna information är tänkt att avhjälpa detta.

8.4 Huvudmenyn

Längst upp i fönstret återfinns huvudmenyn. Den är indelad i fyra undermenyer: `File`, `State`, `Transition` samt `Help`.

- Under `File` återfinns arkivvalen; att skapa ny, öppna, spara eller stänga automater.
- Under `State` återfinns alternativ för tillstånden; lägga till, ta bort och döpa om tillstånd samt val för att sätta som start- eller sluttillstånd och till sist funktioner för att justera tillståndens positioner.
- Under `Transition` återfinns alternativ för transitionerna; lägga till, ta bort och ändra input-tecken samt val för att lägga till och ta bort ε-transitioner.
- Under `Help` återfinns hjälp och information om programmet.

9 AUTLAW - Scenarion

Denna del av implementationen är ägnad åt att granska ett antal typiska användarscenarion. Vad döljer sig egentligen bakom systemet och vilka tänkbara situationer kan tänkas dyka upp för en användare? Svaren på dessa frågor kommer förhoppningsvis att presenteras här.

9.1 Scenario 1: Spåra sträng genom en automat

Det första scenariot är enkelt; användaren är möjligtvis inte van med vare sig programmet eller vid formella språk för den delen och då kan en enkel uppgift till att börja med vara nog så svårt. En tänkbar uppgift är:

"Öppna en existerande automat och spåra en sträng i den"

Hur går eleven till väga och vad händer i systemet?

Öppna en existerande automat

Det bör inte innebära några problem att öppna en existerande automat; detta görs som i andra program genom att välja `File` | `Open` i menyn och i dialogen som kommer upp därefter välja rätt automat och klicka **OK**. Systemet öppnar nu automaten såsom den är sparad och visar innehållet i en ny flik. Samtidigt skapar systemet internt en minimerad deterministisk automat utifrån den öppnade automaten, redo att användas vid jämförelse.

Spåra en sträng

Nu väljer eleven fliken **Test** för att där testa den givna strängen. Strängen skrivs in i textfältet och därefter påbörjas spårningen genom att klicka **Init**. Nu kommer systemt in i ett annat läge, spår-läge, och detta läge bibehålls tills det avbryts med klick på **Stop**. Nu kan eleven tecken för tecken spåra strängen genom automaten, såväl fram- som baklänges genom att klicka på < respektive >. För att testa hela strängen på en gång används >> och för att gå tillbaka till första tecknet används <<.

Resultatet av spårningen presenteras för eleven i textarean under **Test result**.

9.2 Scenario 2: Skapa automat för reguljärt uttryck

Nästa scenario är egentligen tre där svårighetsgraden ökas stegvis. Dessa scenarion är exempel på typiska uppgifter som ingår i exempelvis kurser i kompilatorteknik eller dylikt.

En tänkbar uppgift för en student på en kurs inom formella språk är:

> "Skapa en ändlig automat som svarar mot ett givet reguljärt uttryck".

Hur går eleven till väga och vad kommer att hända i systemet?

Skapa en korrekt automat

Innan eleven själv skapar sin automat behövs en automat att jämföra med elevens. Denna skapas under fliken **Reg.exp.** där eleven matar in det givna reguljära uttrycket. Vilken typ av automat som skapas här är oväsentligt; systemet kommer ändå att konvertera automaten till en minimerad **DFA**för att just kunna jämföra automaten med andra automater. Systemet anropar rutinerna för skapande av automat från reguljära uttryck och automaten skapas under en av flikarna.

Att jämföra automater

När eleven anser sig vara klar med automaten skall denna jämföras med den "korrekta" automaten. Från fliken **Compare** väljs den aktuella automaten och en jämförelse kan påbörjas. Nu skapar systemet automatiskt, dock utan att visa detta för användaren, två stycken minimerade **DFA**er och rutiner för att jämföra de minimerade automaterna anropas. Om systemet nu skulle finna att automaterna ej är ekvivalenta, det vill säga att eleven inte fått till det reguljära uttrycket, ges nu möjlighet för eleven att få exempel på strängar som den ena automaten accepterar och inte den andra.

Att testa strängar

Genom att välja **Test** ges därefter möjlighet att testa den givna exempelsträngen i de båda automaterna. Se mer om detta i Scenario 1 ovan.

Notera
Om detta scenario kan nämnas att det finns två möjligheter att jämföra en automat med ett reguljärt uttryck; i scenariot ovan visas en något "besvärligare" metod (i bemärkelsen att den kräver lite mer jobb), nämligen att skapa en facit-automat utifrån det givna uttrycket. Det finns även möjlighet att direkt under **Compare** jämföra automaten inte bara mot en annan automat utan även mot ett reguljärt uttryck. Fördelen med den beskrivna metoden är att eleven då har möjlighet att spåra en sträng genom såväl den egna automaten som genom facit-automaten.

9.3 Scenario 3: Skapa en DFA från en NFA

En annan tänkbar uppgift för en elev, vilket kan vara lite svårare, är

"Utifrån en existarnde **NFA**, skapa en motsvarande **DFA**."

Hur går eleven till väga och vad kommer att hända i systemet?

Tillvägagångssätt

En elev som skall skapa en **DFA** utifrån en **NFA** börjar med att skapa den automat eleven tror är en **DFA**. Dess deterministiska egenskap kan eleven själv bestämma genom att räkna antalet tillstånd i transitionsfunktionen; för varje tillstånd och varje input-tecken skall finnas exakt ett tillstånd i transitionsfunktionen, ε-transitionen undantagen.

Eleven öppnar nu en facitautomat och jämför denna automat med sin egen automat från fliken **Compare**. Är automaterna ekvivalenta kan eleven sluta sig till att hon gjort rätt; två automater som inte är minimerade behöver nödvändigtvis inte se likadana ut för att acceptera samma språk. Om automaterna inte är ekvivalenta kan eleven erhålla exempel på strängar som skiljer de båda automaterna åt; strängar som sedan kan testas i testfunktionen för att ge eleven information om vad som möjligtvis är fel.

9.4 Scenario 4: Minimera en DFA

Den fjärde och sista uppgiften är det sista steget från omvandling till reguljärt uttryck till en minimerad **DFA**: minimeringssteget. Förutsättning för uppgiften är att en **NFA** omvandlats till en **DFA** och det är denna **DFA** som skall minimeras. Uppgiften är således

"Minimera en existerande **DFA**."

Tillvägagångssätt

Vid den sista uppgiften gör eleven på liknande sätt som tidigare, nämligen att ladda in en facitautomat som jämförs mot den egna. Här är systemet dock mera passivt; man lär komma ihåg att även om eleven inte på ett korrekt sätt minimerat den automat som tidigare gjorts till en **DFA** kommer systemet, som ju omvandlar automaten på ett "korrekt" sätt, tycka att automaterna är ekvivalenta (vilket de givetvis också är eftersom de kommer att acceptera samma språk). Däremot behöver den ena automaten inte vara korrekt minimerad eller minimerad överhuvudtaget (den behöver inte ens vara en **DFA**; systemet sköter i det fallet om konverteringen "i det tysta").

För att avgöra huruvida eleven gjort rätt måste eleven själv jämföra sin minimerade automat med den öppnade facitautomaten. Om och endast om de har exakt lika många tillstånd, och samtidigt är ekvivalenta enligt systemet, är automaten korrekt minimerad. En annan tänkbar möjlighet är att själv välja att konvertera automaten till en minimerad **DFA**. Skulle automaten få färre tillstånd än den tidigare hade kan man sluta sig till att den ej var korrekt minimerad tidigare.

10 Slutsatser och vidare arbete

Vilka slutsatser kan man då dra så här när arbetet är över? Har de mål som sattes upp innan arbetet påbörjades verkligen uppfyllts? Har något kunnat gjorts annorlunda?

10.1 Slutsatser och summering

De mål som sattes upp innan fördjupningen gjordes var att undersöka *vad* som finns att implementera och *hur* detta skulle göras. Det går inte att sticka under stolen med att detta examensarbete förmodligen är mer praktiskt än teoretiskt och det var avgjort redan från början att inga revolutionerande upptäckter skulle göras. Det känns ändå som att de mål som sattes upp har uppfyllts och en liten tillbakablick säger väl ungefär att följande skulle göras:

- Ett system som är inriktat på grundläggande teori om ändliga automater skulle skapas.
- Algoritmer för att konvertera och jämföra automater samt testa strängar skulle undersökas och implementeras.
- Ett användarvänligt och enkelt GUI skulle implementeras för enkel användning av algoritmerna.

Såsom presenterats i denna rapport har de givna algoritmerna tagits fram, analyserats och slutligen implementerats tillsammans med ett GUI som strävar efter att vara så användarvänligt och enkelt som möjligt men ändå så pass kraftfullt att det är användbart i undervisning inom formella språk.

10.2 Vidare arbete

Ett system av denna magnitud blir självklart aldrig helt färdigt även om det system som finns idag fungerar på ett tillfredsställande sätt. Det finns alltid buggar att rätta och ny funktionalitet att lägga till.

Vad som bör hända härnäst inom kort är att skapa någon typ av hemsida för hela arbetet där systemet visas och förklaras och där även en liten online-hjälp finns tillgänglig.

Tittar man på systemet rent funktionellt finns det några saker ytterligare man skulle kunna göra. För att täcka upp hela området reguljära språk skulle systemet behöva

- Funktion för att konvertera en automat till ett reguljärt uttryck.
- Funktioner för att konvertera reguljära grammatiker till automater och vice versa.

Eftersom dessa funktioner inte finns implementerade är inte AUTLAW ett fullfjädrat system för användning inom reguljära språk men det är en god bit på väg.

Vad som ytterligare är tänkbart att utöka systemet med är någon typ av "konverteringsspårare" i vilken användaren får en lite bättre inblick vad som sker när exempelvis en **NFA** omvandlas till en **DFA** och där programmet steg för steg visar vilka tillstånd som slås ihop till ett nytt tillstånd och vilka transitioner som tas hänsyn till och så vidare. Detta skulle förmodligen kunna öka förståelsen för ändliga automater i allmänhet och just konvertering i synnerhet.

Förändringar i GUI

Självklart har även GUIet en del brister och buggar. Vad som speciellt skulle kunna ändras är på det sätt automaterna ritas ut; istället för att ha raka pilar som transitioner kanske man kan tänka sig krökta pilar eller kanske vinklade pilar för att på så sätt enklare kunna placera tillstånden där man önskar ha dem för att öka exempelvis läsbarhet vid skapande av automat utifrån reguljärt uttryck. Någon typ av "intelligens" vid utplacering av tillstånd vid automatiskt skapade automater skulle inte heller vara någonting omöjligt att implementera.

Till sist...

...vill jag passa på att rikta ett ärligt och stort TACK till min handledare Frank Drewes för att han orkat stå ut med mig och för att han aldrig givit upp hoppet, även om jag tror det varit nära många gånger. Flera gånger har han sparkat på mig när jag varit lat men ändå har han alltid sett positivt på det jag gjort och alltid ställt upp och hjälpt mig när jag behövt det som mest. Frank, tusen tack! Du är verkligen en toppenkille!

Sedan vill jag tacka mina vänner som stöttat mig, hjälpt mig eller bara druckit kaffe med mig lite här och var lite nu och då. Speciellt vill jag nämna Martin F, Johan F, Tomas L, Daniel N, Per L och Sabina L men även stort tack till andra andra som på ett eller annat sätt hjälpt mig genomföra och framför allt slutföra detta arbete.

Referenser

[1] Aho, Alfred V., Sethi, Ravi och Ullman, Jeffrey D: *Compilers - Principles, Techniques and Tools* Addison-Wesley (1986)

[2] Hopcroft, John E. och Ullman, Jeffrey D: *Introduction to Automata Theory, Languages and Computation* Addison-Wesley (1979)

[3] Lewis, Harry R. och Papadimitriou, Christos H: *Elements of the Theory of Computation* 2d ed., Prentice-Hall (1998)

[4] Sudkamp, Tomas A: *Languages* and *Machines - An Introduction to the Theory of Computer Science*, 2d ed., Addison-Wesley (1998)

[5] Rodger, Susan H <rodger@cs.duke.edu> *JFLAP 4.0 Beta Version*, 2003-05-22 <http://www.cs.duke.edu/ rodger/tools/jflaptmp/> (2003-06-08)