Регулярни изрази и автомати

Увод

Всеки език, без значение дали е писмен или говорим, е съставен от думи, поради това трябва да имаме основни знания за тях. Разглеждат се компютърни модели за три важни аспекта на думите: техният правопис, произношение и морфология. Очертава се единственият математически модел, който е в основата на всички говорни и езикови процеси: автомата. Съществуват четири вида автомати: крайни автомати (finite-state automata), преобразователи на крайните автомати (finite-state transducers), преобразователи в дълбочина (weighted transducers) и модел на Хидън Марков (Hidden Markov - HMM). Регулярните изрази, включващи пълната дефиниция на Perl са други средства за обясняване и спецификация на думите и дърветата на решенията.

Въведение в регулярните изрази и автомати

Представете си, че сте страстен почитател на рядки видоде горски животни, например еноти. Желанието за много информация ви кара да се обърните към вашият любим браузър и да търсите нещо за тях. Вашият браузър ви връща много страници. Откривате “интересни връзки за еноти и лемури” и всичко за изложенити на опасност видове. Може би предпочитате търсене да се извършва по самата дума “енот” без да се налага да се пише думата в множествено число, т.е. с край “и”. Освен това може да искате да намерите желаните страници независимо дали са използвани главни или малки букви. Може би бихте желали да търсите цени в даден документ като низовете: $199 или $25 или $24.99? В тази глава се въвежда понятието **регулярен израз,** който е стандартното означаване на числа с условни знаци за характеризиране на текстови последователности. Регулярните изрази се използват за определени текстови низове като например търсене в интернет или в други обработващи информация програми, но само играещи важна роля в създаването на думи (в PC, Max или UNIX).

След като дефинираме регулярните изрази, ние ще покажем как те могат да бъдат изпълнени чрез **крайните автомати.** Крайните автомати са

не само математическо средство, но и едно от най-важните средства представящи компютърната лингвистка.

Регулярни изрази

Един от успехите в стандартизацията в компютърната наука са регулярните изрази, език за опреляне на търсене на текстови низове. Регулярните изрази се използва за търсене на тестове в UNIX (Vi, Perl, Emacs), Microsoft Word, Word-Perfect.

Дефиниция

Регулярен израз е низ, представляващ шаблон, който се използва за:

- проверка дали накакъв низ отговаря на някакви условия
- търсене в низ
- извличане на части от низ
- замяна на подниз с друг

Регулярните изрази (първо въведени от Клийни (1965г.)) са формула на специален език, която се използва за определяне на прости класове от низове. Низът е редица от символи. С цел да се създаде техническо усройство за търсене в текст, низът е някаква редица от буквено-цифрени характеристики (букви, номера, пространства и пунктуация). Пространството го представяме като символа “ “.

Формално регулярният израз е алгебрично означение на низове, така те могат да се използват за определяне на търсене, както и за дефиниране на език по формален начин. В началото ще обсъдим регулярните изрази като начин за определяне на търсене в текст. Използването на три регулярни изразни оператори е достатъчно за характеризирането на низове, но ние използваме по-удобен синтаксис на регулярен израз на Perl. Търсенето при регулярните изрази изисква шаблон и множество от текстове**,** които се претърсват. Търсенето при регулярните изрази преглежда множеството от текстове и връща всички текстове свързани с шаблона.

В системата връщаща информация, също както при търсачката, текстовете могат да бъдат цели документи или web страници. При обработването на думи текстовете могат да бъдат индивидуални думи или редове от документ. Когато даваме пример за търсене ще приемаме, че търсачката връща ред от намериния документ. Ще подчертаем частта от примера, която съответства на регулярният израз. Търсенето може да

определи всички съответсвия на регулярният израз или само първото срещнато съответствие.

Основни шаблони на регулярни изрази

Най-простият вид на регулярен израз е последователност от прости знаци. Например при търсенето за “енот”, пишем /енот/. Така регулярният израз /лютиче/ съответства на няколко низа съдържащи подниза “лютиче”. От тук ние можем да сложим разделители около всеки регулярен израз, за да уточним какво е регулярен израз и какво е шаблон. Използваме разделител от както са въведени от Perl, но разделителите не са част от регулярните изрази. Търсеният низ може да се състои от една буква (като /!/) или поседователност от букви (като /urgl/). Първата инстанция на регулярният израз е отбелязана по-долу с почернен шрифт (въпреки че дадено приложение може да избере да връща не само първата инстанция) :

Регулярни изрази	Примерни съответствия
/еноти/	“Интерестни връзки към еноти и лемур”
/а/	“М**а**рия беше спряна от Мона”
/каза Клара/	“Моят подарък, моля,” **каза Клара**
/песен/	“ нашата прекрасна песен”
/!/	“Вие сте нощен крадец !”

Регулярните изрази са чувствителни – те правят разлика между малки и големи букви. Това означава, че шаблонът /еноти/ не съответства на низа Еноти. Можем да разрешим този проблем с използването на квадратни скоби [и]. Низът от знаци вътре в скобите определя различията от знакови съответствия. Следващата таблица показва, че моделът / [еЕ] / съответства на модели съдържащи и “е” и “Е”.

Регулярен израз	Съответствия	Примерни шаблони
/ [еЕ]нот/	Енот или енот	“Енот”
/ [1234567890] /	Някакво число	“Числа от 7 до 5”

Регулярният израз / [1234567890] / определя всяко отделно число. Когато класът от знаци като числа или букви са важни за конструиране на форми в изрази, те могат да бъдат трудни (т. е. трудно е да се определи означението на някоя главна буква). / [А Б В Г Д Е Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ ЙО ЬО Ю Я] /. В тези случаи скобите могат да се използват с тире (-) определящо някой знак от редицата. Моделът / [2-5] / определя един от знаците 2, 3, 4, или 5. Моделът / [а-г] / определя а, б, в или г. Ето и някои примери:

Регулярен израз	Съответствия	Примерни шаблони
/ [А-Я] /	Главни букви	“неговото име е **И**ван”
/ [а-я] /	Малки букви	“**в**али сняг”
/ [0-9] /	Едноцифрени числа	“Глава **1**: Увод”

Квадратните скоби заедно със символа ^ могат да се използват за определяне на това, кой единичен знак не трябва да се съдържа. Ако символът ^ е първият символ след отварящата квадратна скоба [, резултатният модел е отрицание. Например, моделът / [^а] / съответства на единичен знак (включително и специалните знаци) с изключение на а. Това е така само когато символа ^ е първият символ след отварящата квадратна скоба. Ако това се среща другаде, обикновено става въпрос за коректорски знак. Следващата таблица показва няколко примера:

Регулярен израз	Съответствия	Примерни шаблони
[^ А-Я]	Не главни букви	“Н**е**говото име е Иван”
[^о О]	Нито “о” нито “О”	“Вали сняг”
[е ^]	“е” или “^”	“Гладен ^”
а^ б	Моделът “а^ б”	“Виж **а^ б** ”

Използването на квадратни скоби разрешава буквеният проблем за думата еноти. Но все пак нямаме отговор на нашият въпрос дали енот или еноти? Не можем да използваме квадратните скоби, за това използваме питанката /?/, което означава “ знакът с предимство или нищо” няколко примера са показани в следващата таблица:

Регулярен израз	Съответствия	Примерни шаблони
еноти?	енот или еноти	“**енот**”
ка?то	както или като	“**както**”

Можем да мислим за питанката като “нула или една инстанция на предишния знак”. Понякога имаме нужда от регулярни изрази, които ни позволяват повторения. Например разглеждаме езика на овцете, който се състои от низове, които изглеждат по следният начин:

баа!
бааа!
баааа!
ба**а**ааа!
ба**а**ааааа!
……

Този език се състои от низове с буквичката б, продължавайки с буквичката а, следвани от удивителен знак. Друг оператор който ни позволява да казваме “няколко знака от а” е знакът * обикновено наричан Клийни*. Звездата означава “нула или много символа от предишния знак или регулярен израз”. а* означава “няколко низа от нула или многа а-та”. Това ще съответства на а или аааааа. Регулярният израз за съответствие на едно или много а-та е /аа*/, означава “едно а последвано от нула или много а-та”. Изразът / [аб]* / означава “нула или много а-та или б-та”. Това ще съответства на низове аааа или абабаб или ббб.

Знаем достатъчно, за да определим регулярен израз за оценки: сложни числа. Така регулярният израз за цяло число е / [0-9][0-9] */. Понякога имаме да напишем регулярния израз за числа два пъти, това е най-краткият начин да определим “последното”от няколко знака. Това Клийни+, което означава “един или много от предишните знаци”. Този израз / [0-9]+ / е начинът да се определи “поледователност от числа”. Има два начина за определяне на езика на овцата / бааа* ! / или / баа+ ! /.

Много важен специален знак е точката (/ . /) , който означава всеки отделен символ.

Регулярен израз	Съответствия	Примерни шаблони
/ .д /	Някой знак преди “д”	яд, ад

Използва се често заедно със звездата на Клийни * и означава “който и да е низ от знаци”. Например представете си, че в някой ред имаме да намерим определена дума, например aardvark, която се среща два пъти. Можем да определим това с регулярният израз / aardvark . * aardvark /.

Котвите са специални знаци, които “закотвят”. Най-простите котви са символа ^ и символа за долар $. Символът ^ съответства на началото на реда. Шаблонът / ^Аз / съответства на думата Аз ако е в началото на реда. Има три приложения на символа ^ : съответства на началото на реда, като отрицание в квадратни скоби и самото значение на символа. Символът за долар $ съответства на краят на реда. Символът $ се използва за означаване на място в краят на реда и /^Кучето\./$ съответства на ред в, който се съдържа фразата “Кучето”.

Има и още две други котви: \b съответства на гранична дума, докато \B съответства на негранична. Това /\baз\b/ съответства на думата аз, но не и на думата “азбука”. Много технично, Perl определя думи като някаква последователност от числа или букви; това е основно за дефиницията на думи в езиците за програмиране като Perl или C.

Разделяне, групиране и приоритет

Представете си, че имаме нужда от текст за домашни животни: и се интересуваме точно от кучета и котки. В този случай може да искаме да търсим за някакъв низ котка или низ куче. Не можем да използваме квадратните скоби за търсене на “котка или куче”, имаме нужда от нов оператор, разделителен оператор, ще използваме символа | . Моделът / котка | куче / съответства на низа котка или на низа куче.

Понякога се нуждаем да използваме този разделителен оператор в средата на по-дълга последователност. Например, представете си, че искаме да намерим информация за някаква домашна рибка. Как можем да определим guppy (малка тропическа рибка) или guppies? Не можем да кажем / gupp | ies /, защото това ще съответства на guppy и ies. Това е така, защото поледователности като guppy имат предимство над разделителният оператор |. За да се отнася заделителният оператор само за определении изрази имаме нужда да използваме кръгли скоби (и) . Моделът / gupp (y|ies) / определя, че разделянето се отнася само за наставките y и ies.

Кръглите скоби са полезни, когато използваме Клийни*. Оператора Клийни* се отнася само за единични знаци, не за цяла последователност. Представете си, че искаме съответствие на повтаряща инстанция на низ. Имаме ред, който има колони Колона 1, Колона 2, Колона 3. Изразът / Колона [0-9]+ */ няма да съответства на модела от колони. Звездата тук се отнася само за празното място, не за цялата последователност. С кръглите скоби можем да напишем изразът / (Колона [0-9]+ *)*/, който съответства на думата Колона, последвана от число и празно място. Идеята, че един оператор може да има предимство над друг изисква йерархия от приоритет на оператори за регулярни изрази. Следващата таблица дава приоритета на операторите, от най-високият към най-ниският.

Кръгли скоби	( )
Броячи	* + ? {}
Последователност и котви	the ^ my end$

Моделите могат да бъдат двусмиелни. Разглеждаме изразът [а-я]* когато съответствието е за текста “имало едно време”. [а-я]* съответства на “нула или много букви”, на този израз може да не съответства нищо или първата буква и, или им, или има, или имал, или имало. В тези случаи регулярните изрази съответстват винаги на най-дългият низ.

Памет

Кръглите скоби (и) ни позволяват да определяме, че низ или израз може да се срещне два пъти в текст. Например представете си, че искаме да търсим моделът ‘ the Xer they were, the Xer they will be’, където искаме да задължим двата X да бъдат в един и същи низ. Правим това като обграждаме първото X с кръгли скоби и поставяме второто X в числовият оператор /1, както следва: the(.*)er they were the \1er they will be. Тук оператора /1 ще постави низа съответстващ на първия в скобите. Това ще съответства на The bigger they were, the bigger they will be, но не и на The bigger they were, the faster they will be. Числовият оператор може да използва и други числа: \2 означава повторение на целият израз. Например the (.*)er they (.*), the\1er they \2 ще съответства на The bigger they were, the bigger they were, но не и на The bigger they were, the bigger they will be. Тези числови памети се наричат регистри(т. е. регистър 1, регистър 2, регистър 3 и т.н.).

Един прост пример

Предполагаме, че сме искали да напишем регулярен израз, за да намерим случаи на английския пълен член “the”. Прост (но неверен) пример може да е: /the/. Един проблем е, че този образец ще пропусне думата, когато тя е в началото на изречение и тя е главна буква (напр. The). От тук следва примерът: /[tT]he/. Но по този начин се извикват и други думи, в които се съдържа пълният член “the” (напр. other или theology). Така, че е нужно да определим, че искаме инстанции с граници на думата от двете страни, т.е. /\b[tT]he\b/.

Да предположим, че искаме да направим това без използването на /\b/? Може да поискаме това, когато /\b/ няма да третира подчертаванията и числата като гранични области на думата, но може да искаме да намерим “the” в някой контекст, където също може да има подчертавания и числа в съседство (the_ или the25). Нужно е да уточним, че искаме инстанции, в които няма букви от всяка страна на the: /[ˆa-z][tT]he[ˆa-z]/.

Но все още има един проблем с примера: той няма да открие думата “the”, когато е в началото на реда. Тъй като използвахме регулярния израз [^a-z], за да избегнем слятото theS се предполага, че трябва да има някакъв единичен (въпреки това не-буквен) символ преди “the”-то. Може да избегнем това като уточним, че преди “the” трябва да е указано започване на нов ред или не-буквен символ: /(ˆ|[ˆa-z])[tT]he[ˆa-z]/.

Един по-сложен пример

Нека опитаме един по-показателен пример за силата на регулярните изрази. Да предположим, че искаме да изградим приложение, за да помогнем на потребител да си купи компютър по интернет. Потребителят може да иска всякакъв компютър с повече от 233 Mhz и 32MB дисково пространство за по-малко от 1000$. За да направим обратна връзка от този вид първо ще имаме нужда да потърсим изрази като 233Mhz или 32 MB, “Compaq” или “Mac”, 999.99$. В останалата част от тази секция ще разгледаме някои прости регулярни изрази за тази задача.

Първо нека да завършим нашия регулярен израз за цените. Ето регулярен израз за знак долар, последван от низ от цифри. Забележете, че Perl е достатъчно “умен”, за да разбере, че знакът $ тук не значи край на ред: /$[0-9]+/. Сега ще трябва да боравим с дробта на доларите. Ще добавим десетична запетая и две цифри след нея: /$[0-9]+\.[0-9][0-9]/. Този пример позволява само $199.99, но не и $199. Трябва да направим центовете нездължителни, и да се уверим , че сме в границите на думата: /\b$[0-9]+(\.[0-9][0-9])?\b/.

А какво ще кажете за определянето на скоростта на процесора (в Mhz) или за размера на паметта (в наши дни все още в MB)? Тук има няколко примера за това: /\b[0-9]+ *(Mhz|[Mm]egahertz)\b/; /\b[0-9]+ *(MB|[Mm]egabytes?)\b/.

Забележете, че използваме “ ”* за означаване на “нула или повече интервали”. Работейки с дисково пространство (в GB = Gigabytes), имаме нужда отново да използваме незадължителни дроби (“5.5 GB”): /\b[0-9](\.[0-9]+)? *(GB|[Gg]igabytes?)\b/.

Накрая, може да искаме няколко прости примера, за да определим операционната система или марката:/\b(Win|Win95|Win98|WinNT|Windows *(NT|95|98)?)\b/; /\b(Mac|Macintosh|Apple)\b/.

Оператори

В следващата таблица са описани някои специални символи:

^	съвпада с начало на низ
$	съвпада с край на низ
.	съвпада с всеки символ

\ придава или премахва специално значение на следващия символ
\t табулация
\r връщане в началото на реда
\n нов ред
\f долен ред
| разделя алтернативи
() групиране
[] символен клас, съвпада с кой и да е от символите в скобите
[^] ^ в [] обръща значението на символния клас
[-] - в [] дефинира диапазон от символи

предефинирани символни класове

\w символ от дума - буква, цифра или '_' , еквивалент: [a-zA-Z0-9_]
\W обратното на \w
\s празно пространство, еквивалент: [\ \t\r\n\f]
\S обратното на \s
\d цифра, еквивалент: [0-9]
\D обратното на \d

специални символи за количество

“лакоми” версии - съвпадат възможно най-много пъти :
* предният символ или група се повтаря 0 или повече пъти
+ предният символ или група се повтаря 1 или повече пъти
? предният символ или група може и да го няма
{n} предният символ или група се повтаря n пъти
{n,} предният символ или група се повтаря n или повече пъти
{,m} предният символ или група се повтаря m или по-малко пъти
{n,m} предният символ или група се повтаря между n и m пъти

“нелакоми” версии - съвпадат възможно най-малко пъти :
*? , +? , ?? , {n}? , {n,}? , {,m}? , {n,m}?

Следващата таблица показва модификаторите:

i не се прави разлика между малки и големи букви

g	търсенето не спира при първото съвпадение а продължава до края на низа

Таблица с принципи на съвпадането:

Принцип 0: Взет като цяло, всеки регулярен израз ще съвпадне на възможно най-предна позиция в низа.
Принцип 1: От алтернативи a|b|c..., ще бъде използвана най-лявата, позволяваща съвпадане на целия регулярен израз.
Принцип 2: ?, *, + и {n,m} ще съвпаднат с възможно най-голям брой повторения, които все още позволяват целия регулярен израз да съвпадне. ??, *?, +?, {n,m}? ще съвпаднат с възможно най-малък брой повторения, които все още позволяват целия регулярен израз да съвпадне.
Принцип 3: Ако има два или повече елементи в регулярен израз, най-левия (не-)лаком символ за количество, ако съществува, ще съвпадне с възможно най-голяма(най-малка) част от низа, която позволява целия регулярен израз да съвпадне. Следващия най-ляв ще направи същото с остатъка от низа и т.н. докато всички елементи бъдат удовлетворени.

Сравнявайки Клийни* с Клийни+, можем още да използваме определени числа като броячи, като ги заградим с фигурални скоби. Регулярният израз /{ 3 }/ означава “точно 3 появявания на предходния символ или израз”. Също може да бъде определена област от числа, така че / {n, m} / уточнява от n до m появяванията на предишен символ или израз, докато / {n, } / означава поне n появявания на предишен израз.

Крайни автомати (Finite-State Automat)

Сега се обръщаме към теоритичната основа и механизма, който ще използваме за изпълнението на регулярни изрази: крайните автомати (КА). Всеки правилен израз може да бъде изпълнен като краен автомат (с изключение на регулярните изрази, които използват свойствата на паметта). В този параграф ще бъдат представени крайни автомати за някои от изразите от предишния параграф и след това ще бъдат предложени решения как регулярнитете изрази на автоматите преминават в главни, общи изрази. Въпреки че започваме с употребата им за изпълнение на регулярни изрази, крайните автомати могат да се използват и за много други неща.

Използване на крайни автомати за разпознаване на езика на овцата

След време с помоща на папагал докторът се научи да разпознава езика на животните толкова добре, че можеше да им говори и да разбира всичко, което те казват.

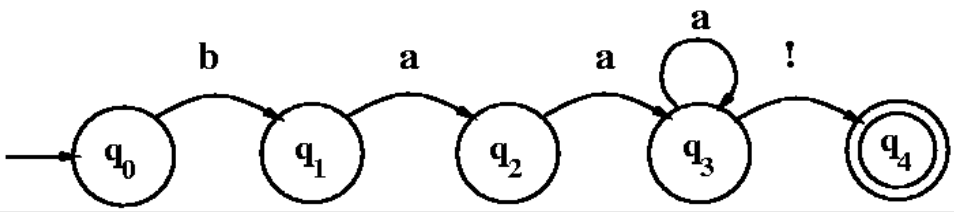
(Хю Лофтинг, Историята на доктор Дулитъл)

Нека да започнем с езика на овцата. Дефенира се като низа:

baa!
baaa!
baaaa!
baaaaa!
baaaaaa!
baaaaaaa!
……

Правилният израз за този низ е baa+!. Фигура 1 показва автомат за моделиране на този регулярен израз.

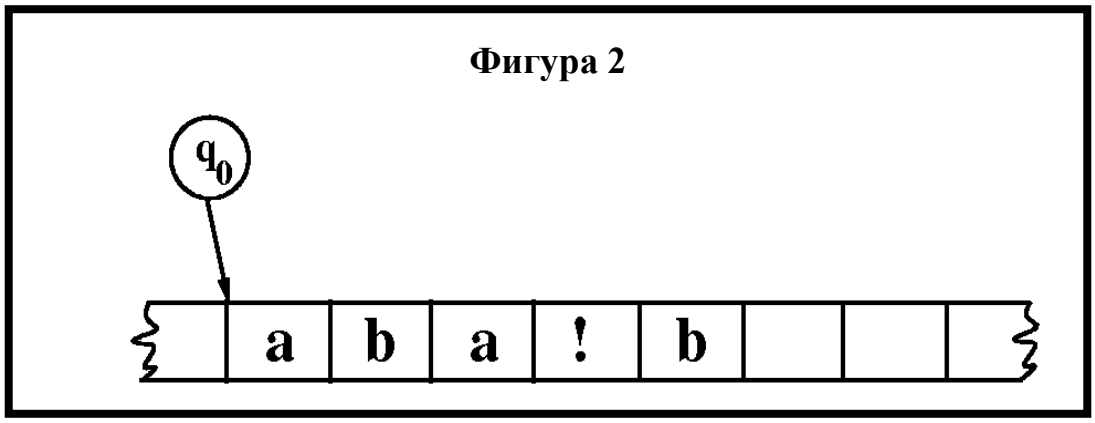
Фигура 1: Краен автомат за езика на овцата.



Автоматът (крайният автомат, FSA) разпознава набор от низове, в този случай низовете характеризират езика на овцата, по същия начин, по който го правят и регулярните изрази. Представянето на автомата е като насочващ граф: крайно множество от пръстеновидно наредени листа (още наречени възли), заедно с множество от насочени връзки между двойка възли, наречени дъги. Ние ще представяме възлите с кръгове, а дъгите със стрелки. Автоматът има пет състояния представени чрез възли в графа. Състоянието 0 е стартовото състояние, което ние представяме чрез входяща стрелка. Състояние 4 е крайното състояние, което ние представяме с две кръгчета. Има още четири прехода, които са представени чрез дъги в графа.

Крайният автомат може да бъде използван за разпознаване (приемане) на низове по следния начин. Първо се мисли за входа като за написана дълга лента начупена на клетки с един символ написан във всяка клетка, както е на фигура 2.

Фигура 2



Механизмът започва в стартово състояние (q_0), повтарящо многократно следния процес: провери следващата буква на входа. Ако е отбелязан символ на дъга напускаща текущото състояние, тогава пресечи дъгата, в противен случай се придвижи в следващото състояние и също провери символа на входа. Ако сме в крайното състояние веднага след като започне входа, механизмът успешно разпознава езикът на овцата. Ако механизмът никога не достигне крайното състояние, или защото е изразходил входовете или се е случило да остане в някое некрайно състояние, казваме че механизмът е отхвърлен или е пропаднал.

Може да представим автомата и с таблица на състоянията. Както в означенията на графа, таблицата на състоянията представя стартово състояние, допустимите състояния и преходите, завършващи със съответните им символи. Следва таблица на състоянията за автомата от фигура 1.

Фигура 3

	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅

Ние отбелязахме състояние 4 с колона, която го определя като крайно състояние (можете да отбележите толкова крайни състояния, колкото желаете), 0 определя нелегален или липсващ преход. Първият ред може да се прочете като: “Ако ние сме в състояние 0 и виждаме входа b, тогава ние преминаваме в състояние 1. Ако сме в състояние 0 и видим входа a или !, тогава пропада прехода.”

По формално краен автомат се дефинира чрез следните пет параметъра:

- Q: множество от N състояния $q_0, q_1, \ldots, q_N$
- ∑: множество от входове (символи на азбуката)
- q_0: началното състояние
- F: набор от крайни състояния, F е подмножество на Q

δ(q,i): преходна функция или преходна матрица между състояния. Даването на състояния qϵQ и вход символа iϵ∑, δ(q,i) връща ново състояние q’ϵQ. δ е така свързано с Q x ∑ и Q;

За автомата за езика на овцата от фигура 1, $Q=\{q_0, q_1, q_2, q_3, q_4\}$, ∑={a,b,!}, $F=\{q_4\}$ и δ(q,i) е дефинирано чрез таблица на състоянията във фигура 3.

Фигура 4 представя алгоритъм за разпознаване на низ, използвайки таблица на състоянията. Алгоритъмът се нарича D-RECOGNIZE за определено разпознаване на детерминиран автомат. Детерминираният алгоритъм е такъв, който няма определени точки; алгоритъмът винаги знае какво да прави при какъвто и да е случай. Следващата част ще представи недетерминиран автомат, който трябва да избира в кое състояние да отиде.

D-RECOGNIZE взима като вход лента или автомат. Връща потвърждение, ако низа посочва мястото на лентата посочено от автомата, а в противен случай пропада. Трябва да се отбележи, че докато D-RECOGNIZE приема той вече посочва кой низ трябва да бъде проверен. Неговата задача е само малка част от главния проблем, който обикновено използва регулярни изрази, проблема е откриване на изрази в корпус (главният проблем е поставен като пример в следваща глава).

D-RECOGNIZE започва инициализирането на променливите начални и текущо състояние (index, current-state) още в началото на лентата и в състоянието на механизма, когато той инициализира. D-RECOGNIZE тогава вписва примка, която движи останалата част от алгоритъма. Той първо проверява дали е достигнат края на този вход. Ако е така, той

приема входа (ако текущото състояние е приемливо) или ако не е така, пропада входа.

Ако са останали други входове на лентата, D-RECOGNIZE проверява в таблицата на преходите, за да реши в кое състояние да премине. Променливата текущо състояние (current-state) определя кой ред на таблицата да бъде прегледан, докато текущия символ на лентата определя коя колона от таблицата да бъде прегледана. Клетката на резултата в таблицата се използва да обнови променливата текущо състояние (current-state) и да се увеличи променливата индекс (index), така че да може да се придвижи напред по лентата. Ако клетката на резултата от таблицата е празна, механизмът приключва.

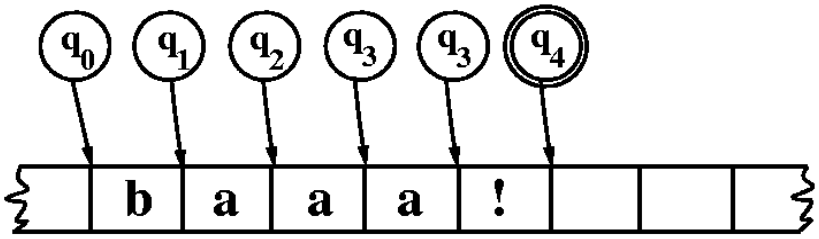
Фигура 4: Алгоритъм, който разпознава крайните автомати. Този алгоритъм връща accept (приемам), ако целият низ е посочен в определения език накрайния автомат и го отстранява, ако низа не се съдържа в езика.

```
function D-RECOGNIZE(tape, machine) return accept or rejest

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
    if End of input has been reached then
      if current-state is an accept state then
        return accept
      else
        return reject
      elsif transition-table[current-state, tape[index]] is empty then
        return reject
      else
        current-state ← transition-table[current-state, tape[index]]
        index ← index+1
    end
```

Фигура 5 очертава изпълнението на алгоритъма за крайният автомат за езика на овцете даващ прост вход baa!

Фигура 5: Очертаване на изпълнението на краен автомат #1 на езика на овцете.



Преди тестването в началото на лентата, механизмът е в състояние q_0. Намирането на b на входа на лентата се променя на състояние q_1, като означение на таблицата на състоянията $[q_0, b]$. След това намира a и се премества на състояние q_2, още едно a го поставя в състояние q_3, трето a го оставя в състояние q_3, когато прочете ! се премества в състояние q_4. Докато няма други входове, състоянието на края на входа в началото на примката е удовлетворено за първи път и механизмът спира в състояние q_4. Състоянието q_4 е приемливо състояние, т.е. механизма е приел низа baaa!, като изречение на езика на овцата.

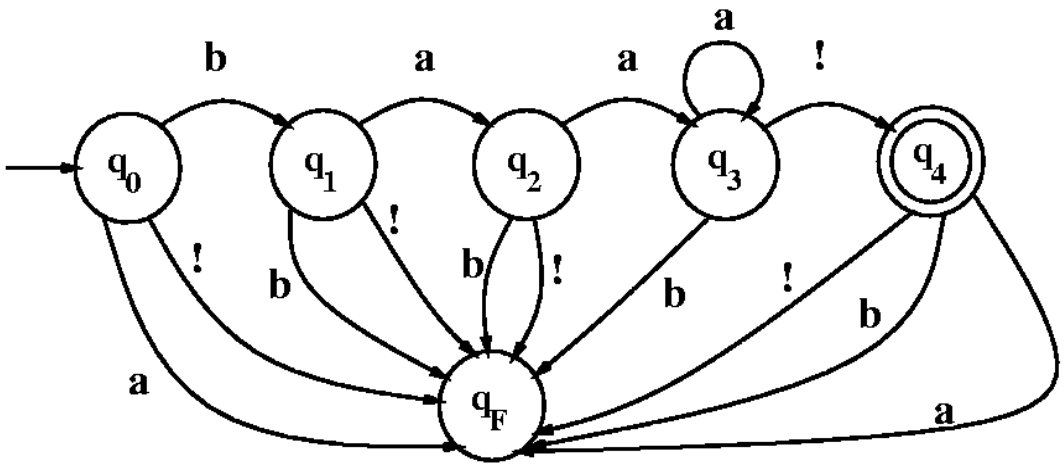
Алгоритъмът ще пропадне, когато няма легален преход за дадена комбинация от състояния и входове. Входът abc няма да може да бъде разпознат докато няма легален преход от състоянието q_0 на входа a (т.е. този вход на таблицата на преходите е 0). Дори автоматът да е позволил първоначално a, то със сигурност ще пропадне на c, докато c не е дори в азбуката. Ние можем да мислим за тези празни елементи в таблицата все едно, че са посочени като празно състояние, което ще наречем пропадащо състояние или хлътнало състояние. Тогава можем да видим, който и да е механизъм с празни преходи, които ние сме увеличили с пропадащото състояние и които рисуваме изцяло с извънредни дъги, така че ние можем да се придвижваме, до което и да е състояние на който и да е възможен вход. За цялостност, фигура 6 показва краен автомат от фигура 1 с пропадащо състояние q_f.

Официални езици

Може да използваме същия граф от фигура 1 като автомат за генериране на езика на овцете. Ако го направим, се казва, че автоматът започва от състояние q_0 и пресича дъгите до новите сътояния отпечатвайки символите, които следват. Когато автоматът достигне до крайното състояние, той спира. Трябва да се отбележи, че в състояние 3, автоматът няма друг избор освен да отпечата ! и да отиде в състояние 4, или да

отпечата a и да се върне в състояние 3. Засега нас не ни интересува как механизмът прави това решение: може би по случайност. Нито ни интересува кой точно низ с езика на овцата ще създадем, докато низът е обхванат от регулярния израз за езика на овцата отдолу.

Фигура 6: Добавяне на пропадащи състояния към фигура 1.



Голяма идея номер 1: *Модел, който може едновременно да генерира и да различава всичко, и само низовете на официалните езици действат като дефиниция на официалните езици.*

Официалният език е набор от низове, всеки низ е съчетание от символи от ограничен набор от символи наречен азбука (същата азбука използвана отгоре за дефиниране на автомата). Азбуката за езика на овцата е зададена: ∑ = a,b,! . Давайки модел m (като точен краен автомат) ние можем да използваме L(m), за да означим "официалния език дефиниран чрез m". Официалният език дефиниран от автомата за езика на овцата m е набора от безкрайности:

L(m) = {baa!, baaa!, baaaa!, baaaaa!, baaaaaa! …}

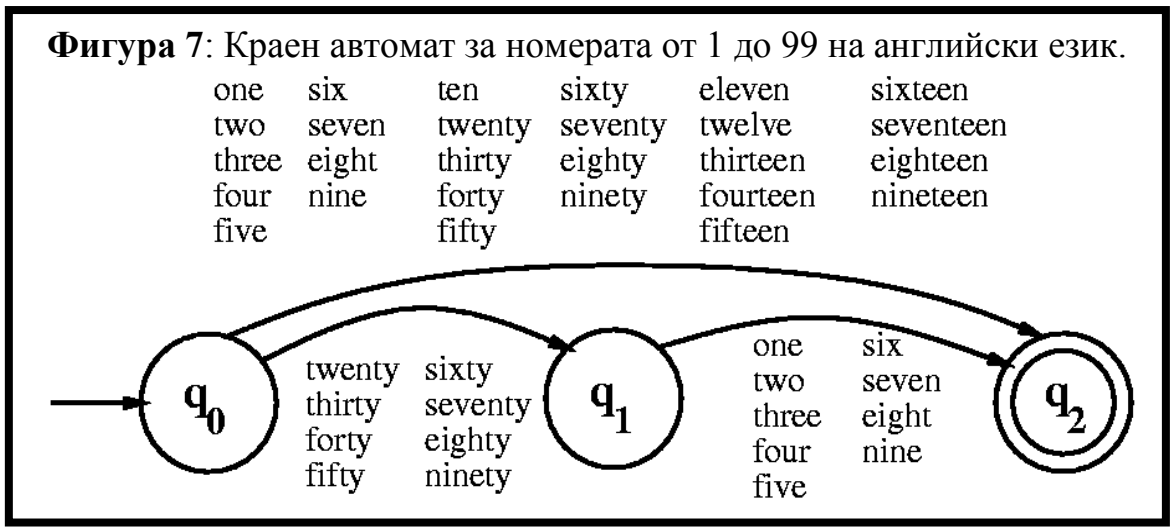
Автоматите са неизползваеми за дефиниране на език, защото те изразяват безкраен набор (като този отгоре) в затворени форми. Официалният език не е същият като естествения език, който е вид от езиците, които хората говорят. Всъщност нормалният език може да няма никаква прилика с реалния език (като пример официалния език може да бъде използван да моделира различни състояния). Но често използваме официалният език за моделиране на част от естествения език, като част от фонетиката, морфологията или синтаксиса. Терминът генеративна граматика понякога се използва в лингвистиката за означаване на

граматика на официалния език; произходът на термина се употребява от автомат за дефиниране на език чрез създаване на всички възможни низове.

Друг пример

В предишния пример нашата официална азбука се състоеше от букви, но ние можем да направим азбука на по-високо ниво състояща се от думи. По този начин можем да напишем краен автомат, който да моделира факти за комбинация от думи. Като пример ще предположим, че искаме да построим краен автомат, който моделира малка част от английски сметки, занимаващи се с пари. Като официален език ще моделираме част от английския език състоящ се от фразите 10 цента, 3 долара, 1долара и 35 цента и т.н.

Започваме първо със създаване на автомат, който трябва да разграничава числата от 1 до 99, когато работим с центове.

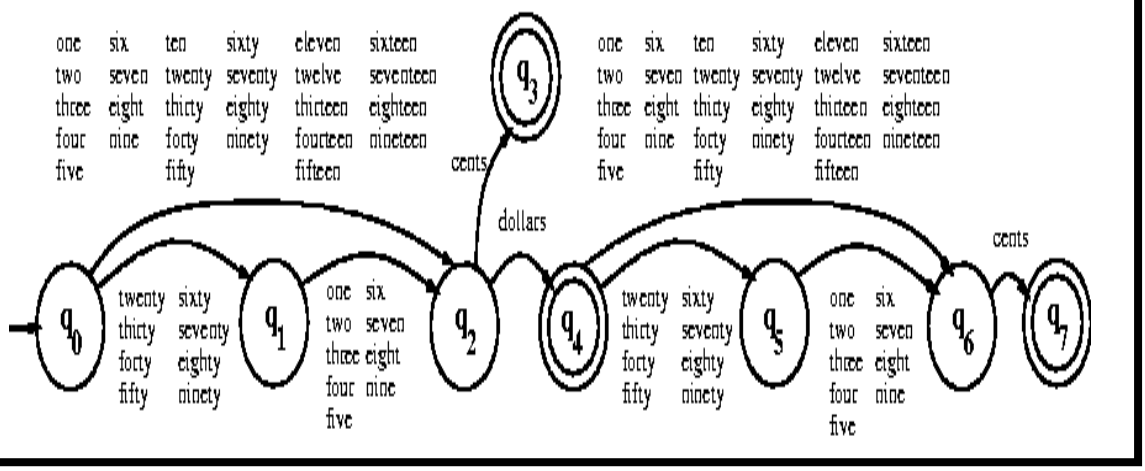


Фигура 7: Краен автомат за номерата от 1 до 99 на английски език.

След това трябва да се въведат долари и центове в автомата. Фигура 8 показва проста версия на това, където са направени две копия на автомата на фигура 7 и са прибавени думите центове и долари.

След това трябва да се въведе в граматиката различни суми от долари; включващи големи числа като стотици и хиляди. Трябва също да направим, така че съществителните като центове и долари да са в единствено число, когато са предназначени за изразяване на един цент и един долар и в множествено число, когато изразяваме 10 цента и 2 долара. Това беше разгледано в предишната глава. За крайният автомат на фигура 7 и 8 може да се мисли като за проста граматика на част от английския език. Ние ще се върнем към създаването на граматики по-късно.

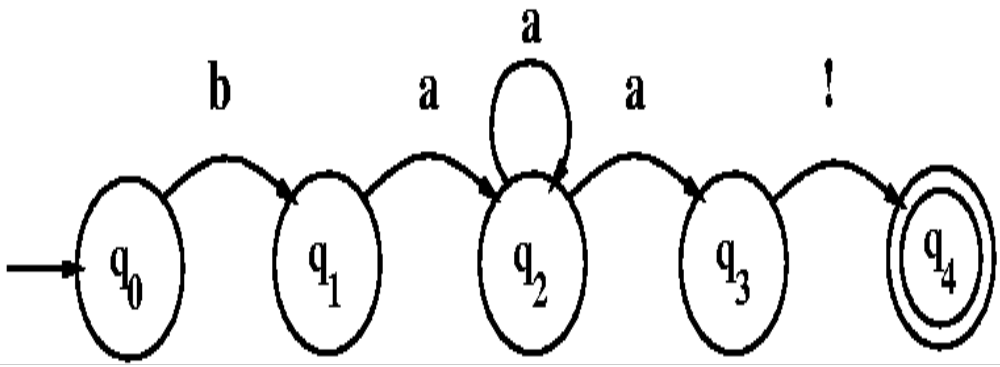
Фигура 8: Краен автомат за долари и центове.



Недетерминирани крайни автомати (НДКА)

Сега ще разгледаме друг клас от крайните автомати – недетерминираните. На фигура 9 е показан НДКА описващ езика на овцата.

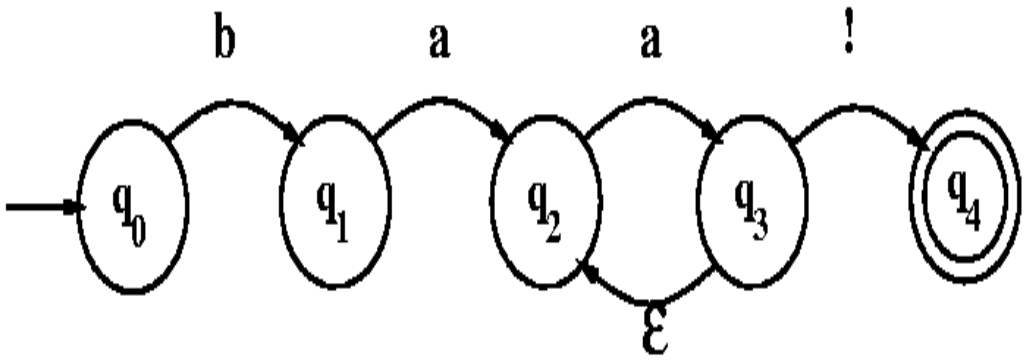
Фигура 9: НДКА за езика на овцата (НДКА #1).



Единствената разлика между този автомат и предишния, е че на фигура 9 примката е при състояние 2 вместо при състояние 3. Когато се достигне състояние 2 не се разбира дали трябва да остане в него или да се продължи към състояние 3. Автомати, които могат да взимат решения се наричат НДКА.

Има друг общ вид от НДКА, който може да бъде описан чрез дъги, които не съдържат символи. Следва пример, който описва езика на овцата с един такъв автомат.

Фигура 10: Друг НДКА за езика на овцата (НДКА #2).



Използване на НДКА за приемане на низове

Ако искаме да разберем дали низа е инстанция на езика на овцата или не и използваме НДКА за разпознаването, трябва да следваме грешна дъга и да я отстраним, когато я открием. Това става когато в едно състояние имаме право на повече от един избор и изберем грешния. Този проблем за избор на недетерминиран модел ще се появява многократно, докато не изчислим модел особено за правене на разбор. Съществуват три стандартни решения на този проблем:

- **Връщане назад:** Когато се достигне до състояние, в което трябва да се направи избор, трябва да се постави маркер, за да се отбележи къде сме били на входа и в какво състояние е бил автомата, така че ако направим грешния избор да можем да се върнем и да опитаме нов път.
- **Гледане във вътрешността:** Трябва да имаме възможност да се проверява във вътрешността, за да си помогнем при избора на път.
- **Паралелизъм:** Когато се достигне до състояние на избор, трябва да се разгледат всичките алтернативни паралелни пътища.

Ще разгледаме първо връщането назад. Този подход предлага правенето на избор да става, по начин, който ни позволява да се връщаме назад до първоначалното състояние при грешен избор. Съществуват два

начина при този подход: трябва да се запомнят всички алтернативни състояния на този избор и трябва да се запази достатъчна информация за тези алтернативи, така че да можем да се върнем към тях при нужда. Когато алгоритъма за връщане назад достигне до състояние, в което не може да се извърши нито един процес (защото е излезнал извън входа или няма легални преходи), той се връща към предишен избор на състояния, който е един от неизследваните до сега и продължава напред.

Фигура 11

	Input			
State	b	a	!	ϵ
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

Прилагайки тази нотация към нашия недетерминиран автомат, трябва да се запомнят две неща за всяка точка на избор: състоянието или възел на автомата, до който ние трябва да достигнем и съответстващата позиция на лентата. Трябва да се извика комбинация от възел и позиция, т.е. търсещото състояние на алгоритъма на автомата. За да се избегне обърквание трябва да се отнесем към състоянието на автомата (т.е. обратното на състоянието на търсене) като възел или като състояние на автомата. Фигура 11 представя такъв вид алгоритъм.

Трябва да се отбележат следните две възможности. Първо: в реда на представяне на възлите, които имат възможност за преход с възврат, се поставя нова колона, която описва точно тези възможности на автомата, в таблицата на преходите. Ако възел има преход с възврат, трябва да се направи списък, в който да се запишат данните за възела в колоната на таблицата с преходи. Второ: трябва да се изчислят съставните преходи до различни възли от един и същи входен символ. Позволява се всяко въвеждане в клетка да съдържа списък от възли с възврат вместо само един възел. Следващата таблицата от фигура 11, показва таблица с преходите за автомата на фигура 9. Докато няма никакви преходи с възврат е показано състояние q_2, а входа a може да доведе обратно до q_2 или q_3.

Фигура 12 показва алгоритъм за използване на НДКА за разпознаване на вход низ. Функцията ND-RECOGNIZE използва променлива (agenda), за да запази информация за всички установени непроучвани избора,

създадени по време на изпълнението на функцията. Всеки избор е съставен от възел на автомата и позиция на лентата. Променливата, която търси в текущото състояние (current-search-state) представя различни избори, които могат да бъдат използвани.

Функцията ND-RECOGNIZE започва чрез създаване на първоначално състояние на търсене и го поставя в променливата – agenda. Засега не е специфицирано, по какъв начин състоянието на търсене се поставя в променливата – agenda. Това състояние на търсенето се състои от първоначално състояние на автомата и указател към началото на лентата. Следва функцията NEXT, която се извиква, за да възстанови отделна точка в списака на променливата – agenda и да я посочи на променливата в текущото състояние (current-search-state).

Първата задача ND-RECOGNIZE на главната примка е да определи дали всички съдържания на ленатата са разпознати успешно. Това става чрез повикване на приемащо състояние (ACCEPT-STATE?), което връща потвърждение, ако текущото търсещо състояние съдържа приемащо машинно състояние и указател към края на лентата. В противен случай машината генерира набор от възможни следващи стъпки чрез извикването на функцията за създаване на нови състояния (GENERATE-NEW-STATES), която създава търсещо състояние, за който и да е преход с възврат и за всеки нормален входен символ от таблицата на преходите. След това всички тези текущи състояния се предават на текущата променлива - agenda.

Следва опит за вземане на ново състояние на търсене, за да се продължи напред. Ако текущата променлива е празна ние изчерпваме възможностите и трябва да отхвърлим входа. В другия случай, когато променливата не е празна се избира една от възможностите и цикъла продължава.

Важно е да се разбере защо ND-RECOGNIZE връща стойност от отхвърлянето само, когато текущата променлива е празна. За разлика от D-RECOGNIZE той не връща охвърляне когато достигне края на лентата в неприемливо машинно състояние. Това е, защото при недетерминирани случаи, такива блокове отбелязват неуспех.

Фигура 12

```
function ND-RECOGNIZE(tape, machine) returns accept or reject
  agenda ←{( Innitial state of machine, beginning of tape)}
  current-search-state ←NEXT(agenda)
loop
  ifACCEPT-STATE? (current-search-state) returns true then
     return accept
  else
     agenda← agendaȕGENERATE-NEW-STATES(current-search-state)
  if agenda is empty then
     return reject
  else
     current-search-state ←NEXT(agenda)
  end
function GENERATE-NEW-STATE(current-state) returns a set of
search-states
  current-node← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
     (transition-table[current-node, e], index)
     ȕ
     (transition-table[current-node, tape [index]], index+1)
function ACCEPT-STATE?(state) returns true or false
  current-state ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state
of
machine then
     return true
  else
     return false
```

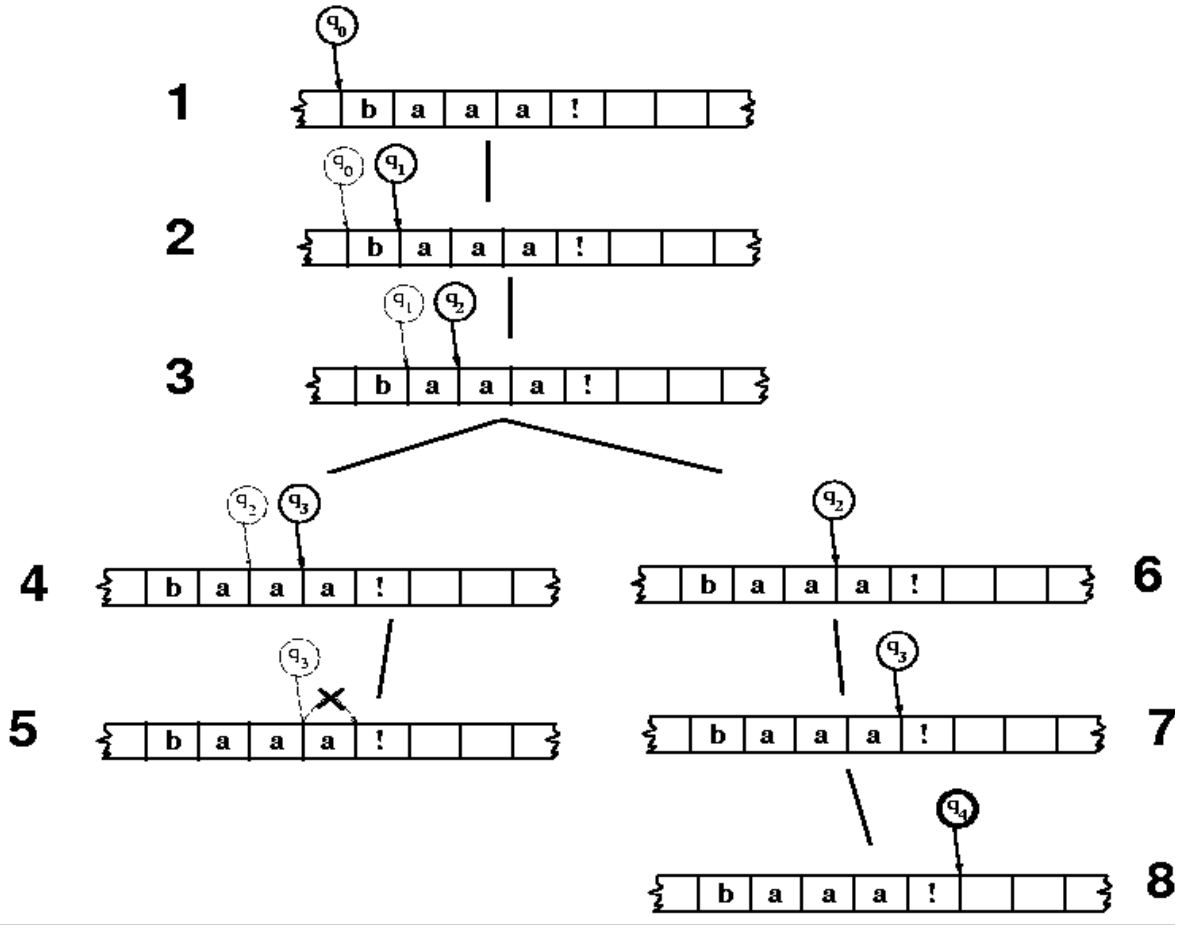
Фигура 13 илюстрира процеса на ND-RECOGNIZE като се опитва да се справи с входа baaa!. Всяка ивица показва състоянието на алгоритъма по зададена точка по време на процеса.

По-интересно става, когато алгоритъма се намира в състояние q_2 докато се гледа второто a на лентата. Проучването на входа a за таблица на преходите $[q_2, a]$ връща и двете състояния q_2 и q_3. Състоянията за търсене

са създадени за всеки от тези избори и са разположени в текущата променлива. За нещастие нашият алгоритмът избира да се премести в състояние $q_{3,}$ преместване, което дава като резултат нито приемливо състояние нито каквото и да е ново състояние докато входа на таблицата на преходите [$q_{3,}$ a] е празен. По този начин алгоритъмът иска от текущата променлива ново състояние за търсене. Докато изборът се върне към q_2 от q_2 е единствения не изследван избор върху текущата променлива и се връща стойност напреднала до a. Доста трудно се намира ND-RECOGNIZE със същия избор. Входът за таблицата на преходите [$q_{2,}$ a] все още посочва връщането назад към q_2 или преминаването към q_3 за валидни избори. Както и преди състоянията, които ги представят са в текущата променлива.

Тези състояния на търсене не са същите като предишните докато стойността на техния индекс не бъде повишена. В същото време текущата променлива agenda достига до следващото преместване на $q_{3,}$ което се отбелязва като следващ ход. Движението до сътоянието и успехът в това състояние е еднозначно определен от лентата и от таблицата на преходите.

Фигура 13: Проследяване на изпълнението на НДКА #1 (Фигура 9) на езика на овцата.



Разпознаването като търсене

ND-RECOGNIZE извършва докрай задачата за разпознаване на низове в регулярните езици, чрез доставяне (използване) на начин за систематично проучване на възможните пътища през машината. Ако това проучване даде път завършващ с приемливо състояние, то низа се приема, в противен случай низа се отхвърля. Това систематично проучване е възможно благодарение на механизма на текущата променлива agenda, който на всяка итерация избира частичен път за проучване и за запазване на следа от всеки останал все още не проучен частичен път.

Алгоритмите като ND-RECOGNIZE, които си служат със системни търсения за решения се наричат state-space search алгоритми. В тези алгоритми, дефиницията на проблема създава пространство от възможни решения; целата е да се проучи това пространство като се връща отговор, когато е намерено такова или се отхвърля входа, когато пространството е изчерпателно проучено. В ND-RECOGNIZE състоянието за търсене съдържа част от машинните състояния с позиции на входната лента. Пространството на състоянията съдържа всички части на машинното състояние и позициите на лентата, които са зададени като въпрос на машината. Целта на търсенето е да се приеме съсътояние, чийто край е на позицията на лентата. Ключът към ефективността на такива програми често е реда, по който състоянията са подредени. Простото нареждане на състоянията може да доведе до изследването на огромни количества безрезултатни състояния докато се намери успешно решение. За нещастие е невъзможно да се различи добрия избор от лошия и често най-доброто, което можем да направим е да се уверим, че всяко възможно решение е обсъдено.

В ND-RECOGNIZE наредбата на състоянията не е определено. Знае се само, че непроучените състояния са включени в текущата променлива agenda, така както са създадени и че функцията NEXT връща непроучено състояние от променливата при поискване на такова. Съществува и вид политика, която може да бъде изпълнена чрез поставяне на новосъздадени състояния пред текущата променлива agenda и поставяне на функцията NEXT за връщане на състояние при поискване от променливата. Това се нарича изпълнение в стек. Обикновенно се извършва търсене първо в дълбочина (depth first search) или стратегията ”последния вътре, първия вън” (Last In First Out).

Тази стратегия търси в пространството следвайки нововъведенията, от както те са създадени. Връща се само, за да обсъди по-ранните възможности, когато прогресира заедно с текущ пример, който е блокиран.

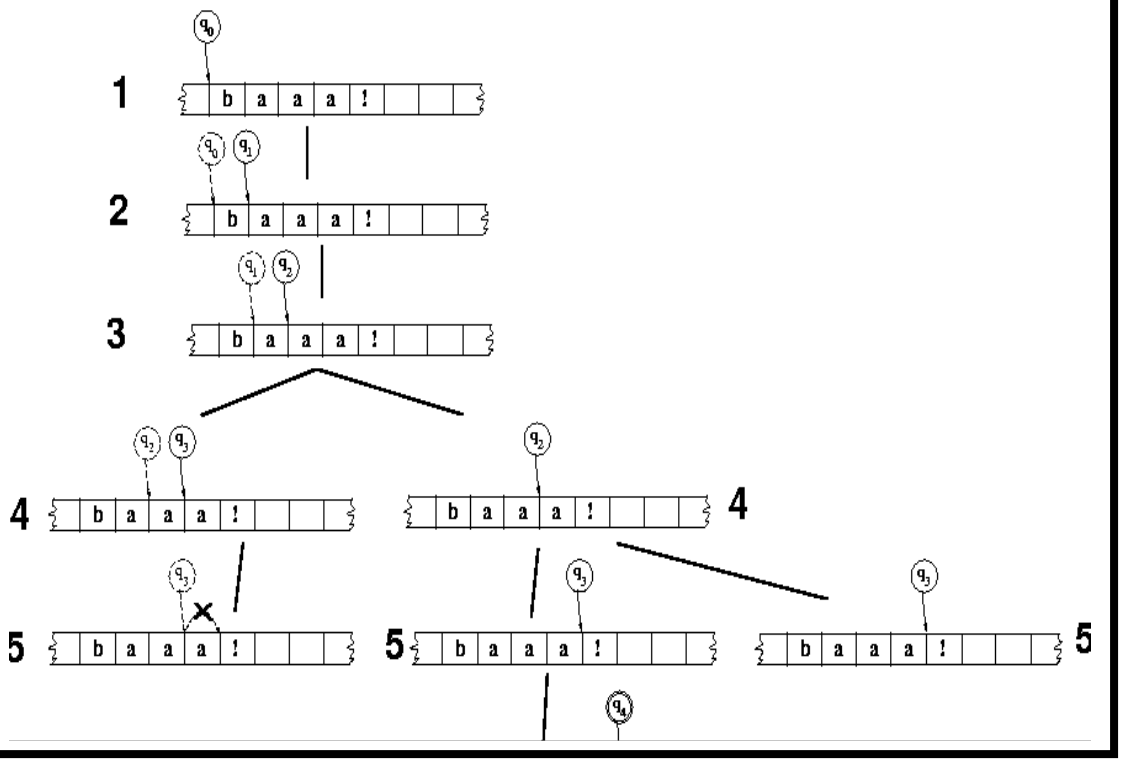
Следата от изпълнението на низа baaa! от ND-RECOGNIZE, която е показана на фигура 13 илюстрира търсене първо в дълбочина. Алгоритъмът попада в първата избрана точка след виждането на ba , когато трябва да реши дали да остане в състояние q_2 или да продължи в q_3. В този случай се избира една алтернатива и тя се следва докато не се установи, че не е грешна, след това се връща обратно нагоре и се проверява слеващата алтернатива.

Сратегията за търсене първо в дълбочина има една голяма клопка: под сигурните обстоятеларства може да се внедри и безкраен цикъл. Това е възможно, ако пространството на търсенето се случи да бъде зададено по такъв начин, че един оператор за търсене може да бъде случайно повторен или да има много голям брой от търсещи оператори.

Втория начин за извикване на оператори в пространството на търсенето е да се обсъдят състоянията по начина, по който са създадени. Този начин на търсене може да бъде изпълнен чрез поставяне на новосъздадените състояния в края на agenda и все още функцията NEXT да връща състояние в началото на agenda. Това изпълнение на agenda се нарича изпълнение при опашка или търсене първо в широчина. Още наречено стратегия “първи вътре първи вън” (First In First Out). На фигура 14 е показано как действа стратегията за търсене първо в широчина. Алгоритъмът започва с първата избрана точка след ba, когато трябва да реши дали да остане в q_2 или да продължи към q_3 . Но сега вместо да се избере един избор и той да се следва, се изследват всички възможни варианти, които са на дървото за търсене.

Като търсенето в дълбочина и търесенето в широчина си има клопки. Като при търсене в дълбочина, ако пространството на състоянията е крайно, търсенето може да не приключи. По-важното е, че при нарастването на размера на текущата променлива agenda, ако пространството на състоянията е дори средно голямо, то търсенето може да изисква прекалено голямо количество памет. За малки проблеми и стратегията за търсене първо в дълбочина, и тази за търсене в широчина, могат да бъдат адекватни, въпреки че обикновенно се предпочита първо в дълбочина поради по-ефикасната употреба на памет. За по-големи проблеми, трябват да бъдат използвани по-сложни техники като динамично програмиране или A*.

Фигура 14: Краен автомат, показващ метода търсене първо в дълбочина.



Свързване на детерминираните и недетерминираните автомати

Ако НДКА имат недетерминирани белези като е-преходи, то това би ги направило много по-силни от КА. В действителност случаят не е такъв; на всеки НДКА съответства КА. Всъщност има прост алгоритъм за превръщане на НДКА в еквивалентен КА, въпреки че броят на състоянията в този еквивалентен детерминиран автомат е много по-голям. Доказателството на съответствието може да се види при Люис и Пападимитроу (1981г.) или Хопкрофт и Улман (1979г.). Основният подход на доказателството е ценен и е изграден по начина, по който НДКА прави синтактичен разбор на входната си информация. Анулирайки различията между НДКА и КА, следва че при НДКА състоянието q_i може да има повече от едно възможно състояние дадено на i-тия вход (напр. q_a и q_b). Алгоритъмът от фигура 12 се справя с този проблем чрез избирането или на q_a или на q_b и ако изборът се окаже грешен, то алгоритъмът приключва. Споменахме, че паралелна версия на алгоритъма би последвала и двата пътя (към q_a и q_b) едновременно.

Алгоритъмът за превръщане на НДКА в КА е подобен на този паралелен алгоритъм. Изградихме автомат, който има детерминиран път за

всеки път, който нашия паралелен разпознавател може да се проследи в пространството на търсенето. Представяме си, че следваме двата пътя едновременно и ги групираме заедно в еквивалентен клас, на който всички състояния се достигат в същите входни символи (напр. q_a и q_b). Сега даваме ново име на състоянието на този нов еквивалентен клас на състояния (напр. q_{ab}). Продължаваме да правим това за всеки възможен вход, за всяка възможна група на състоянията. Резултатният ДКА може да има толкова състояния, колкото има отделения набор от състояния в първичния НДКА. Броят на различните поднабори на набора с N елемента е 2^N, следователно новият НДКА може да има 2^N състояния.

Регулярни езици и крайни автомати

Както предположихме по-горе, класът на езиците, които са формулирани чрез регулярни изрази е точно същият като класът на езиците, които се определят от КА (независимо дали са детерминирани или не). Поради това наричаме тези езици **регулярни езици**. За да дадем точна дефиниция на класа на регулярните езици, трябва да се върнем към две по-ранни концепции: $\sum$ на азбуката, която е набор от всички символи в езика и празния низ "e", който стандартно не е включен в $\sum$. В добавка правим обръщения към празния набор 0 (който е различен от "e"). Класът на регулярните езици (или регулярни набори) обикновено е следния:

1. 0 е регулярен език;
2. всяко a $\in \sum$ U̓ e, където {a} е регулярен език;
3. Ако L_1 и L_2 са регулярни езици, то:
   a) $L_1 . L_2 = \{xy \mid x \in L_1, y \in L_2\}$, конкатенацията на L_1 и L_2
   b) L_1 U̓ L_2, обединението или дизюнкцията на L_1 и L_2
   c) L^*_1, приближението на Клийни на L_1

Всички езици, които отговарят на горните свойства са регулярни езици. Докато регулярните езици са набор от езици, характеризирани от регулярни изрази, в такъв случай всички операции с регулярни изрази представени в тази глава (освен паметта) могат да бъдат изпълнени от три оперции, които дефинират регулярни изрази: слепване, дизюнкция или обединение (също отбелязвано с " | ") и Клийни приближение. Например всички броячи (*, +, {n, m}) са просто специален случай на повторение плюс Клийни*. Всичките котви могат да се приемат като индивидуален специален символ. Квадратните скоби [ ] са вид дизюнкция (напр. [a, b] означава 'a или b', или дизюнкцията на a и b). По този начин е вярно, че

всеки регулярен израз може да бъде превърнат в (може би по-голям) израз, който само налага употребата на трите примитивни операции.

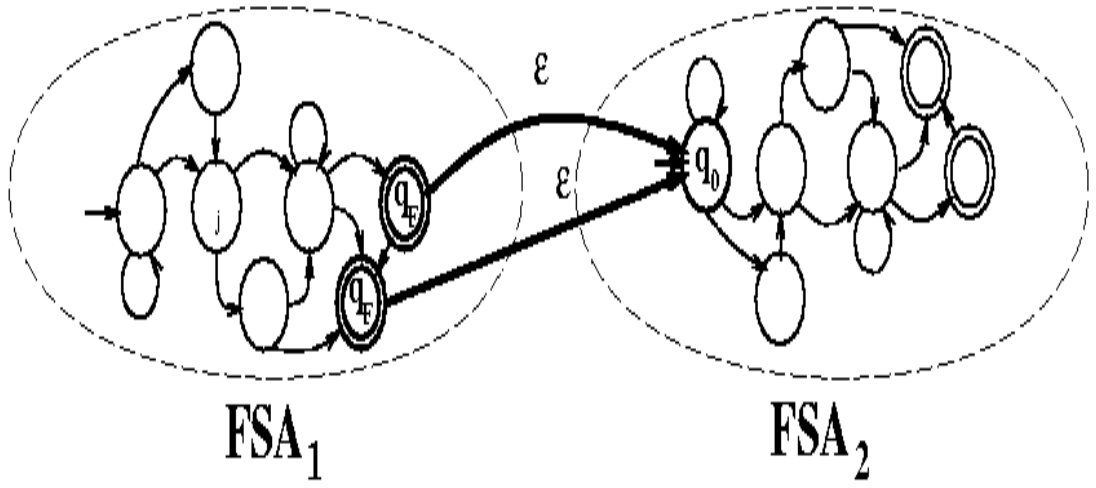
Регулярните езици са също затворени от следващите операции (където E* означава безкраен набор от всички възможни низове, получени от сумата на азбуката):

- Сечение: Ако L_1 и L_2 са регулярни езици, то тогава и $L_1 \cap L_2$ е регулярен език, състоящ се от набора от низовете, които са и в L_1 и в L_2.
- Разлика: Ако L_1 и L_2 са регулярни езици, то тогава и $L_1 - L_2$ е регулярен език, състоящ се от набора от низовете, които са и в L_1, но не и в L_2.
- Допълване: Ако L_1 е са регулярен език, то тогава и $\sum^* - L_1$ е регулярен език, състоящ се от набора от низовете, които не са в L_1.
- Обръщение: Ако L_1 е регулярен език, то тогава и L_1^R е регулярен език, състоящ се от набора от превръщания на всички низове в L_1.

Доказателство, че регулярните изрази са еквивалентни на КА може да се открие при Хопкрофт и Улман (1979г.) и се състои от две части: показва се, че автоматът може да бъде изграден от всеки регулярен език и обратното, че регулярен език може да се изгради от всеки автомат. Няма да даваме доказателството, но ще дадем подхода чрез показване как да се направи първата част: взима се някакъв регулярен израз и се изгражда автомат за него. Подходът е индуктивен: за основен случай може да се изгради автомат за кореспонденция с регулярни изрази от прости символи (напр. изразът a), чрез създаване на начално състояние и приемане на крайно състояние, с дъга между тях именувана a. За индуктивната стъпка показваме, че всяка от примитивните операции на регулярния израз (конкатенация, обединение, приближение) могат да бъдат имитирани чрез автомат:

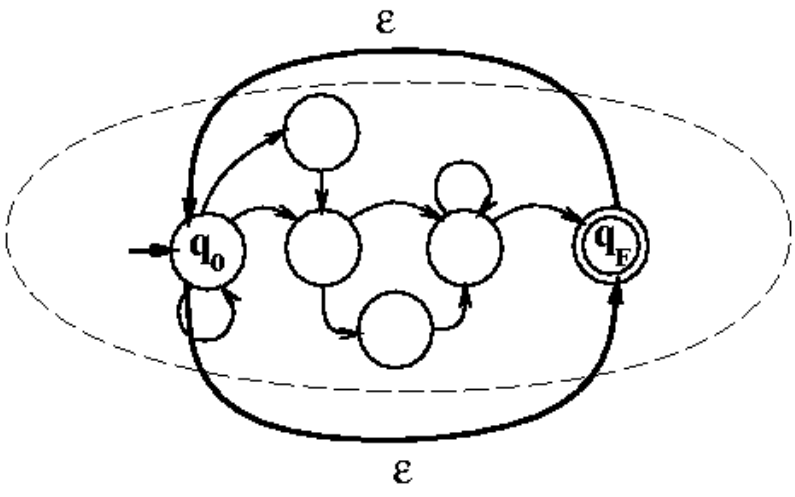
- Конкатенация: Ние просто слепваме два КА един с друг, чрез свързването на всички крайни състояния на КА1 към началните състояния на КА2 чрез е-преходи.

Фигура 15: Конкатенация



- Приближение: Свързваме всички крайни състояния на КА с края на началните състояния чрез е-преходи (това изпълнява частта с повторението на Клийни*) и тогава поставя директни връзки между началните и крайните състояния чрез е-преходи (това осъществява възможността за нулеви появявания).

Фигура 16: Приближение



- Обединение: Добавяме ново единично начално състояние q'_0 и нови преходи от него до всички получени начални състояния на двете машини, за да се присъединят.

Кратко изложение

Тази глава представя по-важните, фундаментални концепции в развитието на езика, автоматизацията и практическите основни средства на автоматизация на регулярните изрази. Ето най-важните точки, които описват тези идеи:

- Езика на правилните изрази е мощно средство за построяване на подобни шаблони.
- Основните операции в правилните изрази включват конкатенация на символи, дизюнкция на символи ([], |, and), броячи (*, +, and{n,m}), връзки (^, $) и приоритета и запомнянето на операциите (\1 and ()).
- Някои регулярни изрази могат да се реализират като крайни автомати.
- Формулиране на безусловен автомат като формален език като се съпоставят подходящи низове на автомат.
- Автомата може да използва някои от символите за неговата лексика включващи букви, думи и дори графични изображения.
- Поведението на ДКА е изцяло детерминирано от неговите състоянията, в които той може да бъде.
- НДКА понякога трябва да прави избор между съставните пътища, да вземе текущото състояние и след това да го постави.
- Всеки НДКА може да бъде превърнат в детерминиран.
- Последователността, по която НДКА избира следващото състояние, което да изследва се определя от стратегията на търсене. Търсенето първо в дълбочина или стратегията Last In First Out съответства на текущата променлива като стек. За да се изследва текущата променлива (agenda) се определя стратегията й за търсене: търсенето първо в широчина или стратегията First In First Out съответсва на текущата променлива като опашка.
- Някои регулярни изрази могат автоматично да се компилират в недетерминирани автомати и след това в детерминирани автомати.

Библиографични и исторически бележки

Крайният автомат възниква през 1950г. В началото модела на Тюринг (1936г.) за компилация на алгоритми се разглежда като основа на

съвременната компютърна наука. Машината на Тюринг е била абстрактна машина с ограничен контрол и входно/изходна лента. В едно преместване, машината на Тюринг ще прочете символ на лентата, записва различния символ на лентата, променя състоянието и се премества от ляво на дясно. (По това машината на Тюринг се различава от автоматите с крайни състояния главно в способността за промяна на символите на лентата.) Втората парадигма е работата на Шанън за информационната теория. Шанън (1948г.) използва състоянието на машините като по същество е идентично с определеното състояние на автомата за модел на свойствата от дискретния информационен канал като телеграф. Шанън също прилага работата си на Марков за процесите наречени "съставните процеси на Марков" за информационния проблем. Съставният проблем на Марков може да бъде видян като автомат с крайни състояния и вероятна промяна между всяка двойка състояния. Шанън обогатява тези модели като някои от промените между състоянията създават символ, така че извежда как може да се извика "модела на Марков на езиците". Третата основа е Мак Калъч-Питс (Мак Калъч и Питс, 1943г.) опростен модел като част от "компютърен елемент", който меже да се опише в термините на логиката на задачата. Този модел е описан двоично като някои точки са активни или не. Вземат се стимуланти и инхибитори, слагат се в други такива методи, ако те активират остарели такива, то те се фиксират още в началото. Базата на Мак Калъч-Питс метода, Клийни (1951г.) и (1956г.) дефинира крайните автомати и регулярните изрази и доказва тяхната еднозначност. НДКА са въведени от Робин и Скот (1959г.), които също доказват тяхната еднозначност за детерминирането им.

Кен Томпсън е един от първите, построил компилирането на регулярните изрази в редактор за текстово изследване (Томпсън, 1968г.). Неговият редактор "ed" включва команда "g/regular expression/p" или отпечатва глобалния регулярен израз, който по-късно удобно се извиква grep.

Съществуват много глобални успехи, внасящи основната математическа теория за автомати; някои личности, които се занимават с това: Хопкрофт и Улман (1979г.) и Люис и Пападимитроу (1981г.). Тези обхващащи математически функции не само простите автомати от този тип, но също и крайните състояния описани в глава 3, свободната контекстна граматика в глава 9, и йерархията на Чомски (глава 13) е много полезно, обширно ръководство за повишаването на използването на регулярните изрази.

Съдържание