Expresii Regulate

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

martie 2000

Subiect: expresii regulate şi aplicaţiile lor

Cunoştinţe necesare: cunoştinţe elementare de programare a calculatoarelor

Cuvinte cheie: expresie regulată, automat finit, analiză lexicală

Cuprins

De la prima întîlnire cu calculatoarele am fost fascinat de ele. Pentru mine întîlnirea asta s-a produs destul de tîrziu, după standardele generaţiei actuale de elevi, prin clasa a opta. Fireşte, primul lucru pe care l-am văzut a fost un joc; calculatorul cu pricina este o specie deja dispărută, numit Sinclair Spectrum ZX, după care românii au făcut o copie destul de reuşită, numită HC 85 (de fapt a existat o a doua copie, TIM-S, dar care s-a bucurat de mai puţin succes). Am avut imediat dorinţa de a construi şi eu un joc.

Realizez că începutul ăsta poate induce în eroare cititorul, pentru că nu despre jocuri vreau să vorbesc aici. De fapt, dacă aţi trecut peste dezamăgirea produsă de titlul anost, cred că veţi rezista şi la restul textului.

De la început am fost fascinat de calculatoare, şi fascinaţia mea a rămas neschimbată, chiar dacă unele dintre motivele fascinaţiei s-au modificat cu timpul. La început am fost entuziasmat de faptul că exista o mică lume asupra căruia sunt demiurg, stăpîn absolut, o lume care se mişcă după legile pe care i le trasez eu. Probabil ca într-o oarecare măsură acest motiv este prezent şi acum în atracţia mea pentru calculatoare, dar într-o măsură mult diminuată.

Cu timpul am învăţat că demiurgul-programator nu este de fapt atotputernic; într-o serie de articole precedente din PC Report, am încercat să ilustrez unele dintre limitările calculatoarelor însele: de exemplu am văzut într-un articol despre teoria complexităţii şi calculabilitate (PC Report din Decembrie 1999, o copie fiind disponibilă din pagina mea de web) că există lucruri pe care calculatoarele nu le pot calcula, şi în trei articole din octombrie, noiembrie şi februarie am încercat să arăt că există probleme care se pot rezolva, dar pentru care oricît de multe resurse computaţionale am avea la dispoziţie, răspunsuri practice probabil nu vor putea fi obţinute vreodată.

Acum calculatoarele sunt profesia mea şi ca atare am de-a face cu ele zi de zi. De fapt studiez calculatoarele de 13 ani, în sensul cel mai concret al cuvîntului: am fost elev la Liceul de Informatică din Bucureşti, student în calculatoare la Politehnica din Bucureşti pentru şase ani (din care unul de master), şi acum sunt în al treilea an de doctorat; cu toate acestea, mai am multe de învăţat! Acum ştiu că de fapt calculatoarele nu sunt un haos primitiv, în care eu, programatorul, atotputernic, pot face ordine, ci că există multe limite peste care nu pot trece, că unele lucruri se pot face mai bine sau mai rău, cîteodată foarte bine şi foarte rău, şi am mai învăţat că sunt enorm de multe lucruri pe care nimeni nu ştie încă să le facă. De fapt asta e o veste bună pentru mine: principala mea activitate este cercetarea, deci faptul că sunt multe lucruri de aflat înseamnă că pentru o vreme o să am ceva pîine pe masă.

Am ajuns pe ocolite la ideea pe care vreau să-mi bazez introducerea. După cum am spus şi în alte articole, ştiinţa calculatoarelor (informatica în terminologie franco-română) are două ramuri mari: partea teoretică şi cea aplicată. În general cercetătorii pot fi atribuiţi destul de clar uneia dintre ramuri. Cei dintr-o ramură uneori îi privesc cu superioritate pe cei din cealaltă. De fapt ambele ramuri sunt extrem de importante, şi deloc uşoare; efortul intelectual depus în oricare dintre ele este substanţial, şi există talent specific fiecăreia dintre ele.

Dar rezultatele cele mai spectaculoase se obţin atunci cînd teoria şi practica concură; situaţia este oarecum asemănătoare cu fizica, în care teoreticienii şi experimentaliştii sunt două triburi separate; dar revoluţiile sunt cele în care teoria şi practica explică un acelaşi fenomen, confirmîndu-se şi întărindu-se reciproc.

De fapt atunci cînd teoria concură cu practica obţinem un al treilea rezultat, cel mai palpabil, şi cel care de fapt schimbă societatea în mod dramatic: este vorba de tehnologie. Cîţi dintre cei care folosesc astăzi televizoarele sunt conştienţi că existenţa lor se bazează pe teoria cîmpului electromagnetic a lui Maxwell, şi pe efectul fotoelectric descoperit experimental de Hertz (şi pentru a cărei explicaţie Einstein a primit premiul Nobel)?

La fel stau lucrurile şi în calculatoare: tehnologia la îndemîna (aproape a) fiecăruia, care acum schimbă societatea într-un mod fundamental (de exemplu prin Internet), nu ar fi fost posibilă fără rezultate atît din teorie cît şi din practică. De fapt acest articol va ilustra o problemă foarte simplă, la care teoria a adus nişte contribuţii majore, dar care are enorm de multe aplicaţii în practică. De fapt, această teorie are un succes atît de mare, încît probabil că nu există calculator digital care să nu o folosească în mai multe feluri, de la cele din PC-uri pînă la microcontrolerele care controlează utilaje industriale sau reglează injecţia automobilelor. Din necesitate însă, prezentarea noastră va fi foarte sumară, şi nu va acoperi decît puţine aspecte ale problemei.

1 Limbaje şi calculatoare

Un calculator abstractizat teoretic poate fi văzut ca un aparat care prelucrează limbaje. Toate informaţiile pe care le prelucrează sunt exprimate în semne discrete, pe care le vom numi "*litere*" (de exemplu în biţi, în zero şi unu, sau dacă preferaţi în octeţi). Astfel, datele de la intrare şi de la ieşire sunt toate şiruri de litere, pe care le numim "*cuvinte*". Noţiunea aceasta de cuvînt este mai largă decît cea obişnuită: în româna nu numim şirul de litere "jjj" un cuvînt, dar în terminologia calculatoarelor, da. Fiecare program procesează anumite cuvinte şi dă ca rezultat alte cuvinte. Adesea, pentru un program nu orice cuvînt venit la intrare are sens: unele vor fi pur şi simplu eronate; de exemplu un program care sortează un şir de numere nu se aşteaptă să primească alte simboluri. Numim o mulţime de cuvinte "*limbaj*".

De exemplu, limbajul Pascal constă în colecţia tuturor cuvintelor care reprezintă programe corecte Pascal (observaţi că un întreg program Pascal este deci un singur cuvînt în această terminologie!). O întrebare este cum poţi descrie în mod succint un întreg limbaj? De exemplu, cum poţi spune cuiva ce este un program Pascal corect? Pentru asta trebuie să foloseşti reguli care descriu cum se pot construi programele corecte. Regulile care descriu cum arată un limbaj se numesc *sintaxa* limbajului.

Aceste reguli la rîndul lor trebuie scrise cumva. Limbajul care descrie regulile poate fi numit "meta-limbaj", pentru că este un limbaj care descrie alte limbaje.

Desigur, ne putem pune întrebarea: cum descriem un meta-limbaj? În mod tradiţional meta-limbajele erau descrise pe hîrtie, folosind o notaţie simbolică; prima probabil în ordine cronologică a fost notaţia Backus-Naur, care a fost printre altele folosită pentru a descrie limbajul Pascal. Un exemplu este prezent în figura 1.

Au apărut apoi programe speciale, care permit descrierea structurii altor limbaje; există multe astfel de scule folosite pe scară largă; iată cîteva dintre ele:

- Una dintre scule, de apariţie recentă, este limbajul XML (eXtensible Meta-Language). XML este un limbaj în care poate fi descrisă sintaxa altor limbaje.

- Generatoare de parsere (ex: `yacc`, `bison`): un parser este un program care verifică apartenenţa unui cuvînt la un limbaj (de exemplu verifică dacă un text dat este un program Pascal). Un

Program Pascal	**Fragment din sintaxa Pascal**

`program ridicol;`
 	
`var a:int, i:int;`
 	
`begin { main }`
`   a = 0;`
`   for i:= 1 to 10 do begin`
`      a = a + i`
`   end;`
`   println a`
`end.`
	Program :=
	   `program` identificator ; bloc .
	bloc :=
	   declaraţii de etichete
	   declaraţii de constante
	   declaraţii de tipuri
	   declaraţii de variabile
	   declaraţii de funcţii şi proceduri
	   `begin` listă de instrucţiuni `end`
	listă de instrucţiuni :=
	   instrucţiune
	      sau
	   instrucţiune ; listă de instrucţiuni

Tabelul 1: Un exemplu de program Pascal şi un fragment (simplificat) din notaţia Backus-Naur, care descrie toate programele Pascal corecte.

generator de parsere este un program care permite descrierea unui parser într-un meta-limbaj. Două programe extrem de folosite în lumea Unix permit descrierea de parsere arbitrare, şi generează cod pentru a implementa aceste parsere în C sau C++; acestea sunt programele `yacc` (Yet Another Compiler-Compiler) şi `bison`, un animal asemănător cu yak-ul[1]. Figura 1 arată cum se foloseşte un astfel de program.

- Chiar în acest articol vom prezenta pe scurt programul `lex`, care este folosit pentru a descrie componentele elementare folosite pentru a scrie programe într-un anumit limbaj.

Pare complicat? Este, cel puţin pînă vă obişnuiţi cu ideea programelor care prelucrează alte programe. Despre XML au mai apărut articole în PC Report; poate în viitor o să scriu un articol despre `yacc` sau prietenul lui. În textul de faţă o să vedem însă soluţia la o problemă mai simplă.

2 Analiza lexicală

Atunci cînd eu vorbesc cu cineva, acel cineva desface textul spus de mine în cuvinte, pe care le analizează. Vorbitorul unei limbi străine are adesea probleme în a *segmenta* textul auzit în cuvinte independente. În scris problema este mai simplă, cel puţin pentru limbile moderne care folosesc alfabetul latin: prin convenţie punem între două cuvinte un spaţiu.

Dacă ideea vi se pare evidentă, aflaţi că lucrurile nu au stat întotdeauna aşa: romanii şi grecii nu foloseau spaţii (de asemenea foloseau extrem de multe prescurtări, pentru că hîrtia era scumpă; din lipsa tipografiilor, încă ne-inventate, copiştii[2] aveau probleme de segmentare, şi făceau

[1] Şi Java are un astfel de generator lexical.
[2] Un fel de Xerox primitiv.

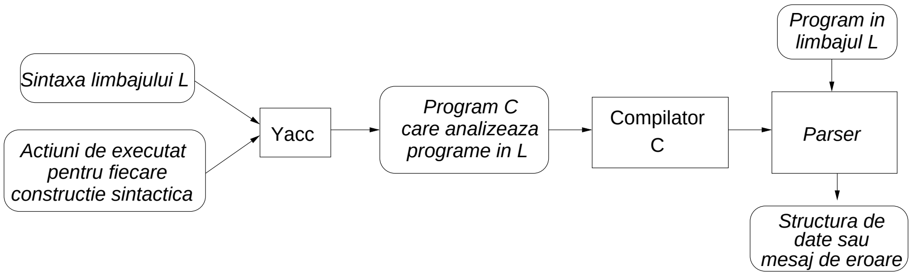

Figura 1: Yacc este un generator de parsere. Utilizatorul descrie sintaxa unui limbaj oarecare L (de exemplu Pascal), folosind o notaţie asemănătoare cu notaţia Backus-Naur; utilizatorul descrie şi o serie de proceduri în C care trebuie să fie executate atunci cînd o anumită construcţie sintactică este întîlnită (de exemplu, ce trebuie făcut cînd întîlnim un `begin-end`, ce trebuie făcut cînd întîlnim un `for`, etc.). `yacc` generează un program C care poate fi compilat şi executat, obţinînd un parser. Acest program apoi primeşte la intrare programe scrise în L şi execută acţiunile indicate. Acţiunile pot descrie de exemplu cum se execută programul, sau cum trebuie construită o structură de date care reprezintă programul (care de obicei apoi este optimizată pentru a face programul mai rapid). Parserul de asemenea recunoaşte programe ilegale, în care caz returnează mesaje de eroare.

adesea erori de interpretare a textelor cînd le copiau). Dacă vi se pare floare la ureche aşa ceva, atunciîncercaţisăcitiţipropoziţiaastaşivedeţicîtdeuşorvăeste.

Atunci cînd unui calculator[3] îi este prezentat un program, el are de făcut o analiză similară: trebuie să despartă textul în bucăţele separate şi să identifice semnificaţia fiecăreia; aceasta este analiza lexicală; bucăţelele obţinute sunt numite *lexeme*. După aceea calculatorul verifică dacă aceste bucăţele sunt îmbinate corect (nu orice succesiune de cuvinte din română formează o propoziţie), folosind analiza sintactică. Figura 2 arată cum aceste faze sunt folosite în compilatoare.

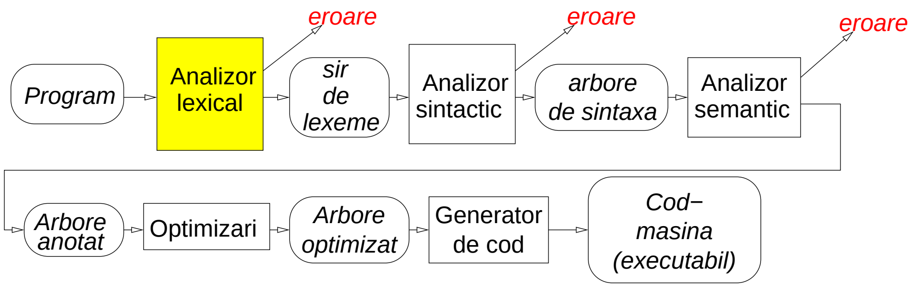

Figura 2: Plasamentul analizei lexicale într-un compilator. Analiza lexicală este prima fază a compilării, care identifică componentele "atomice" ale programului, lexemele. Analiza sintactica a fost descrisă pe scurt în acest text: ea verifică dacă lexemele sunt îmbinate după regulile corecte ale programării în limbajul compilat. Celelalte faze analizează şi optimizează programul, şi nu ne interesează prea mult în acest text.

Nici în calculatoare lucrurile nu au fost prea clare de la început: limbajele primitive, ca Fortran-ul, nu aveau reguli complet ne-ambigue pentru a discerne într-un text care este fiecare cuvînt. De

[3] Mai exact, unui interpretor sau unui compilator.

exemplu, în Fortran spaţiile nu contează (din motive pragmatice: aparatele de găurit cartele nu erau prea fiabile în tastatul de spaţii). Folclorul spune că americanii au pierdut un satelit din cauza asta, pentru că compilatorul de Fortran a interpretat un program altfel decît utilizatorul care l-a scris.

Dacă acestea vi se par aberaţii îndepărtate, aflaţi că nici limbajele cele mai folosite la ora actuală pe scară largă nu sunt lipsite de ele: de exemplu în C++ o problemă specială apare din cauza construcţiei `template`, care se scrie astfel `a<b>`. Problema apare în cazul unor template imbricate (incluse una într-alta), ca în `a<b<c>>`. Problema e cu semnele `>>`: reprezintă ele două template închise, sau semnul "shift-right"? Analizorul lexical nu poate decide de unul singur ce înseamnă aceste semne, are nevoie de ajutor din partea analizorului sintactic.

3 Expresiile regulate: un limbaj pentru a descrie limbaje

Am văzut deci că prima etapă în înţelegerea unui program este descompunerea lui în lexeme. Atunci cînd implementăm un limbaj nou de programare, cum ar fi oare cel mai eficace să descriem toate lexemele posibile? De exemplu, în C avem lexeme de forma `for`, `while`, etc., dar nu lexeme de forma `%$#@`. În plus, într-un program C putem întîlni lexeme de genul `variabila_cea_mare`. Există deci un număr potenţial infinit de lexeme (dacă presupunem că numele de variabile nu au nici o limită pentru lungime). Totalitatea tuturor lexemelor legale este la rîndul ei un limbaj; acesta nu trebuie confundat cu limbajul C: în limbajul C "cuvintele" sunt programele corecte, în limbajul lexemelor C, cuvintele legale sunt toate lexemele care pot apărea în vreun program C.

Teoreticienii au propus cu mult timp în urmă (în anii '60) un meta-limbaj extrem de concis pentru a descrie lexeme. Limbajul acesta este limbajul *expresiilor regulate*. O expresie regulată este un şir de caractere care descrie o mulţime de cuvinte posibile (poate chiar o mulţime infinită). Lexemele tuturor limbajele de programare moderne pot fi descrise prin expresii regulate; vom folosi de aici înainte abrevierea ER pentru "expresii regulate".

Cred că cel mai simplu este să vedem nişte exemple. Pentru început voi folosi notaţia cea mai economică, care este folosită de teoreticieni. Vom vedea apoi tot felul de variante, folosite de tot felul de "scule" de procesare de texte.

Să zicem că vrem să descriem limbaje care folosesc litere din alfabetul {a,b,c,d}, pentru a simplifica lucrurile. Perfect. ER folosesc toate aceste caractere, plus o listă de caractere suplimentare: $\{a, b, c, d, *, \epsilon, \phi, |, \cdot, (,)\}$. Voi explica prin exemple ce înseamnă fiecare. Observaţi că scriu fiecare expresie regulată cu caractere cursive, iar cuvintele din limbajul pe care-l descriem cu caractere drepte. Asta pentru a preveni confuziile, pentru că există caractere comune în cele două mulţimi. (Voi abandona această convenţie în secţiunile care descriu "sculele", pentru că acestea oricum folosesc alte notaţii.)

- Avem în primul rînd 4 ER extrem de simple, care constau dintr-un singur caracter, una din litere. De exemplu, expresia regulată "a". Aceasta este o expresie foarte simplă, care descrie un limbaj format dintr-un singur cuvînt, care are o singură literă; limbajul "{a}".

- Avem apoi expresia regulată ϕ, care descrie limbajul...care nu are nici un cuvînt! Ce risipă! Cui îi trebuie astfel de limbaje? Cu toate acestea, această expresie regulată se dovedeşte cîteodată utilă, chiar dacă majoritatea din programele care manipulează ER nu o folosesc.

- O altă expresie regulată este ϵ. Aceasta descrie un limbaj în care există un singur cuvînt, dar care cuvînt nu are nici o literă! Altă prostie, nu? Această expresie regulată este însă foarte utilă.

- Parantezele le vom folosi ca în matematică, pentru a indica căror expresii li se aplică anumite operaţii. Aşa cum $a*(b+c)$ arată că întîi facem adunarea şi apoi înmulţirea, expresia $R\cdot(S|T)$ arată că întîi aplicăm operaţia | şi abia apoi pe $\cdot$.

  În fine, să ne apucăm de ceva serios. Celelalte trei semne care au rămas ne permit să creăm limbaje mai complicate din limbaje simple. Astea vor face ER o sculă foarte puternică.

- Semnul |, citit *alternanţă*, se pune între alte două ER. Dacă R şi S sunt astfel de expresii, putem scrie $R|S$. Dacă R descrie un limbaj R, iar S descrie un limbaj S, atunci $R|S$ descrie limbajul R $\cup$ S, adică limbajul format din toate cuvintele care sunt fie în R fie în S.

  De exemplu, expresia regulată $a|b|\epsilon$[4] descrie un limbaj cu trei cuvinte: { , a, b }. Primul cuvînt este cuvîntul fără nici o literă; mai avem apoi alte două cuvinte, de cîte o literă. De îndată ce vom vedea celelalte două operaţii o să apreciaţi mai tare semnul |, pentru că împreună ele putem exprima limbaje din ce în ce mai interesante.

- Semnul $\cdot$, numit *concatenare*, se foloseşte la fel ca |, între alte două ER. Fie R limbajul descris de R şi S limbajul descris de S. Atunci $R \cdot S$ este limbajul în care fiecare cuvînt este compus dintr-un cuvînt din R ataşat în faţa unui cuvînt din S. Interesant, nu?

  De exemplu, expresia regulată $(a|b)\cdot(a|c)$ descrie un limbaj compus din patru cuvinte: {aa, ac, ba, bc}. Fiecare cuvînt este compus dintr-un cuvînt din limbajul $(a|b)$ ={a,b} concatenat cu un cuvînt din limbajul $(a|c)$ ={a,c}.

  Deja putem folosi ER pentru a ne exprima foarte succint. De pildă, dacă limbajul R are 2 cuvinte, atunci $R\cdot R\cdot R\cdot R$ are $2\times 2\times 2\times 2 = 16$ cuvinte!

- În fine, semnul $*$, numit şi operaţia “star”, este singurul semn care ne permite să descriem limbaje infinite! Operaţia $*$ se aplică unei singure ER, şi se scrie astfel: R^*.

  Limbajul R^* este un limbaj ale cărui cuvinte sunt formate dintr-un număr arbitrar (inclusiv zero!) de cuvinte din R, concatenate. De exemplu, limbajul descris de a^* este { , a, aa, aaa, aaaa,…}, care conţine cuvîntul fără nici o literă (obţinut din zero copii ale lui “a”), cuvîntul a, cuvîntul aa, etc., şi nu vă puteţi aştepta să scriu aici toate cuvintele, pentru că sunt în număr infinit, şi PC Report recomandă o limită de 3500 de cuvinte pe articol (pe care oricum o să o depăşesc, mi-e teamă).

3.1 Exemple de ER

Iată deci că fiecare expresie regulată descrie un întreg limbaj, poate chiar infinit. Vom vedea aici mai multe exemple de expresii regulate interesante. De acum înainte voi folosi tot setul de caractere ASCII, şi nu mă voi mai limita la literele a, b, c, d. Pentru eficienţă voi denumi ER care apar şi pe care vreau să le refolosesc; voi folosi semnul egal =; acesta nu este un semn care poate fi folosit pentru a scrie expresii regulate.

[4] Operaţia | este asociativă, aşa că o putem folosi fără paranteze.

$CF = 0|1|2|3|4|5|6|7|8|9$. Această expresie regulată descrie cifrele. Nu e mare brînză.

$NN = CF \cdot (CF^*)$. Aceasta este deja o expresie mai interesantă, care descrie toate numerele naturale în notaţie zecimală: cel puţin o cifră, urmată de oricîte alte cifre. Facem convenţia că operaţia $*$ are prioritate mai mare decît $\cdot$, aşa că putem scrie expresia precedentă şi ca $CF \cdot CF^*$.

$NZ = (\epsilon|-) \cdot NN$. Această expresie descrie toate numerele întregi, pozitive sau negative: un număr precedat de nimic sau de semnul minus.

$CH = CF|a|b|c|d|e|f$. Această expresie regulată descrie toate cifrele hexazecimale.

$HX = 0 \cdot x \cdot CH \cdot CH^*$. Acestea sunt toate numerele hexazecimale pozitive în notaţia acceptată de limbajul C.

$FP = (\epsilon|-) \cdot ((CF^* \cdot (.|\epsilon) \cdot CF \cdot CF^*)|CF \cdot CF^* \cdot . \cdot CF^*)$ Această expresie descrie toate numerele reale cu un număr finit de zecimale, (în limba engleză, cu punct în loc de virgulă). Expresia a complicată pentru că permite ca fie partea întreagă, fie partea zecimală să lipsească, dar nu amîndouă.

LIT =literele mari şi mici; mi-e lene să scriu expresia asta în detaliu, seamănă cu CF.

$ID = LIT \cdot (LIT|CF)^*$: identificatorii (numele de variabile) acceptabili într-un program Pascal: o literă, urmată de litere şi cifre.

$b^* \cdot a \cdot (b^* \cdot a \cdot b^* \cdot a)^*$: toate cuvintele formate din literele a şi b în care apar un număr impar de a-uri. Expresia asta am format-o astfel: ceva b-uri (poate niciunul), cel puţin un a, apoi un număr par de a-uri, cu ceva b-uri între ele.

Pentru cei doritori de formalisme, există tot felul de reguli care ne permit să simplificăm ER complicate. Iată cîteva dintre ele (semnificaţia semnului egal este: limbajul descris de expresia din stînga este acelaşi cu limbajul descris de expresia din dreapta):

$$
\begin{aligned}
R|R &= R \\
R|S &= S|R \\
(R|S)|R &= R|(S|T) \\
(R \cdot S) \cdot T &= R \cdot (S \cdot T) \\
R \cdot \epsilon = \epsilon \cdot R &= R \\
R \cdot \phi = \phi \cdot R &= \phi \\
R|\phi &= R \\
R \cdot (S|T) &= (R \cdot S)|(R \cdot T) \\
(R|S) \cdot T &= (R \cdot T)|(S \cdot T) \\
\epsilon|RR^* &= R^* \\
RR^* &= R^*R
\end{aligned}
$$

$$
\begin{aligned}
R^{**} &= R^* \\
R^* \cdot R^* &= R^* \\
R \cdot (S \cdot R)^* &= (R \cdot S)^* \cdot R
\end{aligned}
$$

Primele nouă din regulile acestea ne spun că mulţimea tuturor cuvintelor împreună cu aceste operaţiile $|$ şi $\cdot$ este un semi-inel idempotent. De aici un amator de algebră poate deriva tot felul de alte proprietăţi interesante, pe care acum o să le trecem cu vederea.

3.2 Limbaje regulate şi limbaje neregulate

Am văzut tot felul de ER, şi o să mai vedem şi alte cîteva. Cu cîteva semne simple putem descrie într-un mod finit limbaje complicate infinite. Dar nu orice limbaj se poate descrie cu ER. Limbajele care se pot descrie cu ER se numesc *limbaje regulate*. Puterea ER este destul de limitată, dar suficient de mare pentru a le face extrem de utile. Vom vedea mai jos că sculele care folosesc ER adaugă o serie de abrevieri utile, şi că uneori extind ER în moduri care permit descrierea unor limbaje care nu sunt de fapt regulate.

Iată nişte exemple de limbaje care *nu* se pot descrie cu ER:

- Limbajul "parantezelor potrivite", care constă din şiruri de caractere "(" şi ")" astfel încît parantezele se închid în mod legal.
- Limbajul tuturor programelor Pascal.
- Limbajele formate din cuvinte care sunt palindroame (se pot citi la fel de la cap la coadă şi de la coadă la cap).
- Limbajul cuvintelor care sunt concatenarea a două cuvinte identice.

3.3 Aplicaţii ale limbajelor regulate

Ce facem cu o expresie regulată? Problema principală este cea de *recunoaştere*: dacă avem o expresie regulată şi un text, reprezintă acel text un cuvînt din limbajul descris de expresia regulată? Spunem că expresia regulată *acceptă* acel cuvînt.

De exemplu, cuvîntul "141231" este în limbajul descris de expresia regulată NZ de mai sus, dar cuvîntul "−+−+131" nu este.

O problemă înrudită este cea a *analizei lexicale*, pe care am descris-o mai sus: dacă se dă un text (de exemplu în Pascal) şi mai multe ER (de exemplu ID, NN, etc.), să-l descompunem într-o secvenţă de cuvinte astfel încît fiecare cuvînt este acceptat de una din expresii. De exemplu textul `acum := 51;` este compus din patru cuvinte: un identificator, acceptat de ID, semnul de atribuire, acceptat de expresia $: \cdot =$, pe care nu am descris-o, numărul 51, acceptat de expresia NN şi semnul punct-şi-virgulă, care este acceptat de o expresie regulată foarte simplă, ";".

De fapt cele două probleme de mai sus (acceptarea şi analiza lexicală) se pot ambele rezolva cu aceeaşi tehnologie: dacă putem recunoaşte un cuvînt, atunci putem ciopîrţi şi textul în bucăţi, recunoscînd primul cuvînt, şi după aia continuînd descompunerea cu restul textului. Vestea cea bună este că putem rezolva problema recunoaşterii limbajelor regulate în mod foarte eficient!

4 Automate finite

Aparatele care recunosc limbaje regulate se numesc *automate finite*. Putem vedea aceste automate sub forma unor dispozitive fizice, reale, sau sub forma unor programe extrem de simple.

Teoreticienii arată (şi studenţii învaţă la cursurile de teoria limbajelor formale) că o expresie regulată se poate traduce într-un automat finit, care operează asupra unui cuvînt, şi care produce un rezultat da/nu, dacă cuvîntul este sau nu în limbajul descris de expresia regulată. În acest articol nu o să ne preocupăm despre cum se face asta, şi nici despre tot felul de alte rafinamente, (cum sunt automatele finite nedeterministe). O să vă arăt doar un automat finit şi să vă explic cum funcţionează el. Vom observa că automatul finit se uită la fiecare literă din cuvîntul de recunoscut o singură dată, şi face o singură operaţie internă pentru fiecare literă. Asta înseamnă că procesează fiecare cuvînt în *timp linear*, ceea ce îl face foarte eficient.

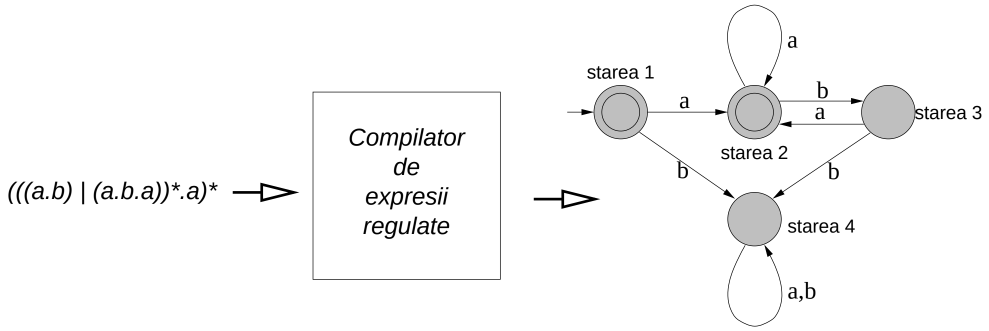

Figura 3: Compilarea unei ER într-un automat finit. Orice expresie regulată E poate fi transformată într-un automat finit care recunoaşte orice cuvînt din limbajul descris de E, şi doar acele cuvinte. Starea 1 este starea iniţială, iar stările 1 şi 2 sunt stări finale.

De fapt aşa procedează toate sculele pe care le expun mai jos: cînd primesc o expresie regulată, "compilează" această expresie într-un automat finit. Automatul este reprezentat sub forma unui progrămel foarte rapid. Apoi acest progrămel primeşte cuvîntul de testat, îl studiază, şi produce diagnosticul. Automatele folosite în analizoarele lexicale indică cum programul de la intrare poate fi descompus în lexeme, şi ce expresie regulată s-a potrivit cu fiecare lexemă.

Figura 3 arată un automat finit care operează cu cuvinte peste alfabetul {a,b}. Acest automat are o mulţime finită de *stări*, notate prin cercuri. În fiecare clipă automatul se află într-o unică astfel de stare. La începutul calculului automatul se află în starea marcată de o săgeată care vine "de nicăieri".

Între două stări există săgeţi etichetate cu litere; acestea se numesc *tranziţii*. De exemplu, între starea 1 şi starea 2 există o tranziţie etichetată cu a: asta înseamnă că automatul, dacă vede litera a la intrare şi se află în starea 1, va trece în starea 2.

Unele dintre stările automatului sunt marcate cu două cercuri; astfel de stări sunt numite *stări finale*. Un cuvînt este prin definiţie acceptat dacă atunci cînd este prezentat la intrare cauzează automatul să treacă într-o stare finală atunci cînd cuvîntul a fost în întregime prelucrat.

De exemplu, figura 4 arată evoluţia în timp a automatului anterior.

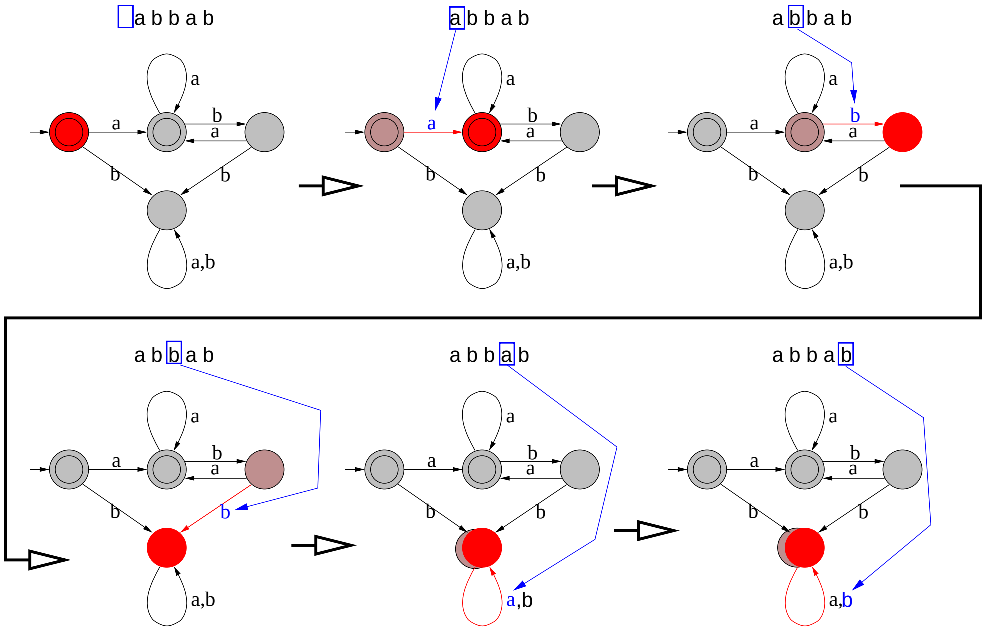

Figura 4: Execuţia automatului din figura 3 cînd la intrare este prezentat cuvîntul “abbab”. Automatul face cinci paşi, cîte unul pentru fiecare literă. Starea curentă este indicată cu roşu, starea anterioară cu roz, şi poziţia curentă în cuvîntul de la intrare printr-un dreptunghi albastru. Săgeţile albastre indică cum este “consumat” cuvîntul de la intrare. Pentru că după terminarea şirului automatul nu este într-o stare finală, tragem concluzia că acest cuvînt nu e descris de expresia regulata din figura 3.

5 Programe care manipulează ER

În această secţiune voi discuta pe scurt mai multe programe care folosesc ER. Fiecare din ele are notaţii puţin diferite de celelalte, ceea ce e foarte neplăcut dacă doriţi să folosiţi mai multe dintre ele.

Toate aceste programe prelucrează fişiere text, adică fişiere în care informaţia este într-o formă citibilă de către oameni. E o tradiţie în lumea Unix ca datele să fie menţinute în formă textuală; Microsoft aparent şi-a făcut un scop să creeze formate binare (adică ne-textuale) de stocare a datelor, care mai sunt şi secrete pe deasupra! Din cauza asta, multe din aceste scule sunt de utilitate redusă în lumea Windows. Din fericire, una dintre formele cele mai prevalente de prezentare a informaţiei este limbajul HTML, care are o reprezentare pur textuală. Pentru prelucrări de HTML, prelucrarea textelor de programe, programe automate de colectat informaţii de pe Internet, sculele descrise de mai jos sunt extrem de utile.

În acest text eu am folosit litere cursive pentru a scrie expresiile regulate, şi litere drepte pentru cuvinte. Din păcate în lumea programelor se foloseşte un acelaşi set de caractere pentru a denota literele şi expresiile regulate. Asta ridică o problemă suplimentară, pentru că nu avem noi semne, $*, \cdot, |, (,)$ pentru a construi expresiile regulate, deci trebuie să folosim tot unele dintre litere. Dar atunci e o ambiguitate între litera ***** şi semnul * folosit pentru expresii regulate.

Toate programele de mai jos rezolvă această problemă în acelaşi fel, şi anume folosesc mai multe simboluri consecutive pentru una dintre semnificaţii, şi un singur simbol pentru cealaltă. De exemplu, primul program, `grep`, utilizează semnul `*` pentru asterisc (operaţia) şi `\*` pentru caracterul steluţă.

5.1 Shell-ul Unix

Am scris în iunie 1997 un articol despre shell-ul din Unix (mai curînd ca să explic ce este, decît ca să arăt cum se foloseşte un shell modern sofisticat). Pe scurt, shell-ul este un program care poate fi folosit pentru a da comenzi sistemului, de obicei în mod interactiv.

Shell-ul Unix tradiţional recunoaşte nişte forme extrem de primitive de expresii regulate, care sunt abreviate într-un mod nu tocmai natural. Pe de altă parte shell-ul permite doar forme restrînse de expresii, aşa că voi începe cu el, pentru că e mai simplu.

Următoarele construcţii în shell sunt pe post de expresii regulate:

- Orice caracter este o expresie regulată; excepţii sunt semnele de mai jos.
- Semnul "?" ţine locul unui singur caracter, oricare ar fi el. Pentru a scrie chiar caracterul semn de întrebare trebuie să scriem `\?`.
- Pentru a scrie semnul `\` trebuie să scriem `\\`.
- Semnul de concatenare este omis. Aşa cum în matematică cîteodată omitem semnul de înmulţire, în shell nu scriem $\cdot$. Astfel, şirul `a???b` reprezintă toate cuvintele de cinci litere care încep cu "a" şi se termină cu "b".
- Semnul "*" nu mai e un operator care se aplică unei expresii; el înseamnă un şir oarecare de zero sau mai multe caractere. Astfel, cînd scriem "`a*b`" asta înseamnă: toate şirurile care încep cu "a" şi se termină cu "b".
- Alternanţa se reprezintă în felul următor: în loc de $R|S|T$ se scrie `{R,S,T}`. Astfel, expresia `{a,b}{c,d}` reprezintă patru şiruri: ac, bc, ad, bd.

În mod normal atunci cînd shell-ul primeşte o comandă de la utilizator, încearcă să expandeze expresiile regulate în toate numele de fişiere de pe disc care se potrivesc. Astfel comanda: `ls .??*` va afişa (comanda `ls`) toate fişierele al căror nume începe cu semnul punct şi are cel puţin trei caractere.

Chiar şi MS-DOS aparent oferea astfel de expresii, dar de fapt era mult mai limitat: puteai avea cel mult o steluţă pentru numele fişierului şi una pentru extensie. În Unix poţi spune ceva de genul `rm *a*b*`, care înseamnă "şterge toate fişierele care au un a şi un b în nume, în ordinea asta". Deşi expresia `*a*b*` pare mult mai complicată, teoria automatelor finite ne spune că ea poate fi analizată şi folosită la fel de rapid ca orice expresie mai simplă.

În Unix poţi spune chiar lucruri de genul: `wc -l */*/*.cc`, pentru a număra liniile (`wc -l` = word count lines) din toate fişierele C++ aflate la două directoare adîncime.

5.2 `grep`

`grep` este un utilitar extrem de util, care a fost proiectat special pentru a manipula expresii regulate. Numele său vine de la Get Regular Expression Patterns, adică "caută textele descrise de o anumită expresie regulată". `grep` primeşte o expresie regulată şi o listă de fişiere, şi caută în acele fişiere liniile care conţin cuvinte descrise de acea expresie regulată.

Există mai multe specialităţi de `grep`, şi cel mai comun dintre ele are tot felul de opţiuni (de genul: tipăreşte numerele de linie, ignoră diferenţa între literele mari şi mici, tipăreşte numai ce *nu* se potriveşte, etc.). Noi vom studia doar funcţionalitatea de bază.

`grep` oferă o paletă foarte largă de operaţii pentru a descrie expresiile regulate, care se dovedesc a fi foarte utile. Pentru că unele semne sunt atît litere cît şi operatori în ER, regulile sunt cam încîlcite; le voi prezenta aici pe cele mai importante:

- Semnul `.` (punct) ţine locul oricărui caracter, mai puţin sfîrşitul de linie. De exemplu, expresia "`...`" descrie orice cuvînt de trei litere.
- Semnul de concatenare se omite; ex. `a.b` înseamnă orice cuvînt de trei litere care începe cu "a" şi se termină cu "b".
- Parantezele se scriu astfel: `\(` şi `\)`, şi se pot folosi pentru a schimba ordinea de aplicare a operaţiilor.
- Semnul de alternanţă este `\|`, ex. `\(a\|b\)c` reprezintă cuvintele ac şi bc.
- Operatorul star se scrie chiar ca o steluţă: ex. `(ab)*`.
- Fiecare caracter, mai puţin cele speciale, este o expresie regulată.
- Unele caractere speciale se prefixează cu `\` pentru a fi exprimate; de exemplu, `\*` reprezintă chiar o steluţă.
- Putem scrie `[a-f]` pentru a indica toate caracterele între a şi f, inclusiv (a,b,c,d,e,f).
- Putem scrie `[^a-f]` pentru toate caracterele care *nu* sunt între a şi f.
- Putem scrie `[acfz]` pentru unul dintre caracterele a, c, f sau z.
- Semnul `^` este începutul de linie.
- Semnul `$` este sfîrşitul de linie.
- Semnul `\?` se aplică ca steluţa, după o expresie, şi înseamnă: "expresia anterioară de zero sau una ori". Cu alte cuvinte, expresia `ab\(c\?\)` reprezintă cuvintele ab şi abc.
- Putem spune: expresia R repetată de două, trei sau patru ori, astfel: `R\{2,4\}`. De exemplu, expresia `\(ab\)\{1,3\}` descrie cuvintele ab, abab şi ababab.

Dacă invocăm `grep` astfel: `grep` ER `fisier`, atunci se întîmplă următoarele lucruri:

1. `grep` transformă automat expresia regulată ER în `^.*`ER`.*$`.

2. Expresia rezultată obţinută este compilată într-un automat finit.

3. Fişierul/ele indicate sunt parcurse, şi fiecare linie care este acceptată de automatul finit este tipărită.

De ce schimbă `grep` expresia regulată în acest fel? Prin definiţie, `grep` tipăreşte toate liniile care *conţin* un cuvînt care se potriveşte cu expresia regulată. Faptul că linia conţine un cuvînt potrivit este acelaşi lucru cu faptul că linia are nişte caractere oarecare, cuvîntul şi apoi alte caractere. Ceea ce e totuna cu a zice că linia însăşi se potriveşte cu expresia regulată expusă mai sus!

Notaţia asta pare complicată, şi chiar este. `grep` este însă o sculă de neînlocuit, pe care eu o folosesc zilnic.

În realitate, problema este şi mai complicată: adesea tastăm comanda grep chiar în linia de comandă a shell-ului. De aceea, shell-ul, înainte de a executa comanda, face anumite substituţii în linia de comandă, şi abia apoi pasează programului `grep` argumentele.

De exemplu, dacă tastăm în shell `grep a\* fisier`, shell-ul va interpreta caracterele `\*` drept semnul steluţă (protejată pentru a nu fi interpretată ca o expresie regulată, ci ca un caracter), şi ca atare va trimite drept argumente lui `grep` doar şirul `a*`, care pentru grep înseamnă altceva decît `a\*`.

Această complicaţie suplimentară face programarea în shell cu expresii regulate o treabă foarte neplăcută. Există însă o soluţie simplă: întotdeauna înveliţi argumentele lui `grep` cu ghilimele simple `''`, şi atunci shell-ul nu se va mai atinge de ele. Veţi scrie deci `grep 'a\*' fisier`.

Dacă sunteţi amator de expresii încîlcite, puteţi scrie şi `grep a\\\* fisier`, pentru că shell-ul va trimite lui `grep` din `\\` doar un `\`, şi din `\*` doar steluţa.

5.3 awk

`awk` este un limbaj de programare foarte simpatic, creat de trei inşi celebri: Aho, Weinberger şi Kernighan (ultimul autor a creat şi limbajul C); iniţialele celor trei au dat şi numele limbajului. `awk` seamănă puţin cu `grep`, în sensul că procesează fiecare linie din fişier separat. Spre deosebire de `grep`, `awk` este un limbaj complet de programare, cu variabile, bucle, etc.

În prezent limbajul `awk` a fost subsumat complet de Perl, care este mult mai eficient, aşa că nu-l voi mai discuta aici.

5.4 sed

`sed` înseamnă Stream EDitor, adică un editor de texte ne-interactiv. În `sed` "bagi" un fişier şi un program cu reguli de "re-scriere". `sed` parcurge fişierul de obicei linie cu linie şi îl rescrie conform cu aceste reguli.

Ca şi `awk`, `sed` este complet surclasat de limbajul Perl. Voi indica aici doar una din trăsăturile lui, care este foarte folosită în Perl: comanda `s///`. Această comandă are forma `s/ER/inlocuire/`. `s` vine de la *substituţie*. Expresiile regulate sunt foarte asemănătoare cu cele ale lui `grep`.

Iată algoritmul după care operează `sed`:

1. Fiecare linie din textul de la intrare este comparată cu expresia regulată indicată.

2. Fiecare linie care se potriveşte setează nişte variabile.

3. Fiecare linie care se potriveşte este apoi înlocuită conform cu `inlocuire`. `inlocuire` poate folosi variabilele setate. Liniile care nu se potrivesc rămîn neschimbate.

4. Linia rezultată este tipărită şi ciclul continuă cu următoarea linie.

Iată un exemplu, pentru clarificare: `sed 's/\([^ ]*\)[ ]*\(.*\)/\2 \1#/' fisier`. Asta se citeşte aşa:

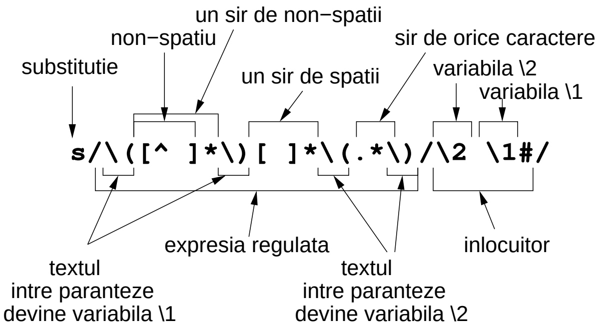

Figura 5: Cum interpretează `sed` comanda din textul articolului.

- Caută să găseşti de la începutul liniei (`^`) o secvenţă de caractere care nu sunt spaţii (`[^ ]*`).
- Aceste caractere vor deveni prima variabilă (lucru indicat de parantezele `\( \)`).
- Apoi trebuie să urmeze un număr de spaţii `[ ]*`.
- Apoi urmează un număr oarecare de caractere `.*`.
- Aceste ultime caractere devin variabila a două (`\( \)`).
- După ce o linie se potriveşte, ea este transformată astfel: `\2 \1#`. Adică a doua variabilă este pusă la început, urmată de un spaţiu, urmată de prima, urmată de semnul `#`.

Dacă aplicăm acest program următorului fişier:

```
Acesta este un mic
fisier pe care incercam
un mic program sed
```

obţinem

```
este un mic Acesta#
pe care incercam fisier#
mic program sed un#
```

Am mutat astfel primul din linie cuvînt la sfîrşitul liniei!

5.5 vi

vi este un editor de texte primitiv, probabil primul editor de texte vizual (numele înseamnă VIsual editor). vi a fost conceput şi implementat de Bill Joy, care este unul dintre fondatorii companiei Sun Microsystems, pe vremea cînd era încă student la Universitatea din California Berkeley.

Cu toate că este relativ primitiv, vi oferă o serie de prelucrări cu expresii regulate extrem de puternice, care într-o anumită măsură nu mai sunt oferite de nici un editor actual!

Expresiile regulate şi comenzile în vi sunt foarte asemănătoare cu cele din sed. Iată doar un exemplu:

```
:/Introducere/,/Capitolul/s/<I>\(.*\)<\/I>/<B>\1<\/B>/g
```

Această comandă complicată trebuie citită după cum urmează:

1. :/Introducere/,/Capitolul/ Pentru toate liniile între cea care conţine cuvîntul “Introducere” şi cea care conţine cuvîntul “Capitolul” aplică comanda care urmează fiecărei linii.

2. Comanda este o substituţie s

3. /<I>\(.*\)<\/I> se citeşte astfel: orice şir de forma <I>ceva caractere</I>[5] este salvat în variabila 1, lucru indicat de paranteze \(\).

4. Apoi porţiunea care se potriveşte este înlocuită cu <B>\1<\/B>, adică: păstrează textul neschimbat (variabila \1) însă înlocuieşte I-ul cu un B[6].

Figura 2 arată un exemplu de text înainte şi după prelucrare.

Inainte	**După**
Un text <I>inainte</I> de introducere. Introducere Un text in <I>interiorul</I> <B>introducerii</B>. Foarte <I>scurt</I>. Capitolul 1 Un text in <I>interiorul</I> capitolului 1.	Un text <I>inainte</I> de introducere. Introducere Un text in <B>interiorul</B> <B>introducerii</B>. Foarte <B>scurt</B>. Capitolul 1 Un text in <I>interiorul</I> capitolului 1.

Tabelul 2: Modificările făcute asupra unui text de comanda vi din text.

[5] <I> în HTML descrie cuvinte scrise cu litere cursive (I = italic).
[6] În HTML <B> indică caractere grase (B = bold).

5.6 emacs

Editorul de texte `emacs` este un editor non-wyswyg extrem de puternic, şi editorul meu favorit[7]; acest text a fost scris cu `emacs`; Ca şi `vi`, `emacs` oferă expresii regulate puternice în comenzi de genul search-and-replace. Nu le voi discuta aici, însă; sintaxa şi expresivitatea sunt foarte asemănătoare cu `vi`.

5.7 f?lex

`lex` şi `flex` sunt două generatoare de analizoare lexicale pentru construcţia de compilatoare. `flex` este o generaţie mai nouă. Aceste programe primesc un fişier cu descrieri de expresii regulate, şi cu acţiuni asociate fiecărei expresii. Apoi aceste programe generează programe C sau C++ (există şi `jflex` pentru Java mai nou) care implementează automatele finite care recunosc aceste expresii regulate, şi execută acţiunile indicate de fiecare dată cînd întîlnesc una din ele.

Dar aceste programe se pot folosi şi în alte scopuri pentru prelucrarea de texte; de pildă eu am scris un program în întregime în `flex` care translatează texte scrise într-un subset al limbajului LaTeX(un limbaj pentru typesetting — aranjare în pagină) în text chior. Eu îmi scriu toate articolele pentru PC Report în LaTeX, dar le transform în text înainte de a le trimite revistei.

Iată un exemplu de program simplu scris în `lex` adună toate numerele naturale dintr-un fişier, indiferent unde apar ele:

```
%{
#include <stdio.h>
#include <stdlib.h>

int suma=0;
%}

CIFRA   [0-9]
NUMAR   {CIFRA}+
%%

{NUMAR} { suma += atoi(yytext); }
.       /* ignora orice nu e un numar */
%%
int yywrap() { return 1;}

int main()
{
    yylex();
    printf("Suma %d\n", suma);
    return 0;
}
```

[7]Am scris un articol întreg în PC Report despre el în mai 1997.

5.8 Perl

În fine, voi încheia cu o sumară prezentare a expresiilor regulate în Perl. PC Report a publicat cel puţin un articol despre acest interesant limbaj, şi poate va continua seria.

Perl vine de la Practical Extraction and Report Language: un limbaj pentru extras date şi produs rapoarte. Este un limbaj minunat pentru a procesa fişiere text şi a face transformări sofisticate. Pînă de curînd era limbajul preferat pentru a genera pagini dinamice la serverele de web, deşi în ultima vreme are concurenţi serioşi în limbaje şi sisteme ca Python, PHP, Visual Basic, Zope, etc.

Perl este departe de a fi un limbaj ideal, dar repară cu succes problemele expresiilor regulate din celelalte limbaje şi scule în mod normal folosite. Perl are următoarele avantaje:

- Oferă toate capabilităţile celorlalte sisteme; în mod intenţionat integrează tot ce pot face `awk`, shell-ul, `grep` şi `sed`.
- Oferă o sintaxă unică pentru a descrie expresiile regulate; fiecare din programele de mai sus are reguli puţin diferite de celelalte.
- Oferă o gamă foarte amplă de operatori cu expresii regulate. Foarte multe abrevieri practice sunt pre-definite.
- Perl oferă chiar o extensie care de fapt *nu este* o expresie regulată, în sensul teoretic al cuvîntului, pentru că poate fi folosită pentru a descrie limbaje neregulate.
- Cînd scrieţi programe Perl nu mai aveţi problema că semnele scrise sunt interpretate întîi de shell şi abia după aia de program. În felul acesta, ce scrieţi, asta ajunge la compilatorul Perl.
- Perl poate prelucra cuvinte care se întind pe mai multe linii (celelalte programe, mai puţin `f?lex` se limitează la o singură linie de text pentru fiecare expresie regulată!).
- În fine, sintaxa Perl este mult mai omogenă. De exemplu, niciodată nu se pune semnul \ în faţa unui operator pentru expresii regulate. Dacă puneţi acest semn, caracterul devine un caracter obişnuit, şi nu un operator.

Voi menţiona o construcţie Perl care depăşeşte puterea expresiilor regulate. Din această cauză, algoritmii eficienţi cunoscuţi, care genereazā automate finite, nu mai funcţionează pentru astfel de expresii. Ele trebuie deci folosite cu cumpătare, pentru că anumite cazuri patologice vor necesita extrem de mult timp.

Perl ne permite să scriem ceva de genul: `(.*)\1`; asta înseamnă un şir de caractere urmat de el însuşi, deci un cuvînt care este format din concatenarea a două şiruri identice. Această exprimare simplifică multe operaţii, dar nu este o ER în sensul strict al cuvîntului, deşi manualul Perl o numeşte aşa.

Închei acest paragraf cu o listă a operatorilor şi prescurtărilor cele mai importante din Perl (nu toate!). ER din Perl sunt extrem de bogate, dar învăţate încetul cu încetul sunt foarte abordabile. Dacă faceţi multe prelucrări pe fişiere, Perl este o sculă absolut necesară în arsenalul dumneavoastră.

<table>
<tr><th>Expresie</th><th>Semnificaţie</th></tr>
<tr><td><code>\</code></td><td>Următorul meta-caracter devine un caracter obişnuit</td></tr>
<tr><td><code>^</code></td><td>Început de linie</td></tr>
<tr><td><code>.</code></td><td>Orice caracter în afară de newline</td></tr>
<tr><td><code>$</code></td><td>Sfîrşit de linie</td></tr>
<tr><td><code>|</code></td><td>Alternanţă $(a|b)$</td></tr>
<tr><td><code>()</code></td><td>Indică precedenţa operaţiilor</td></tr>
<tr><td></td><td>Concatenarea este implicită (nu folosim nici un semn)</td></tr>
<tr><td><code>[]</code></td><td>O mulţime de caractere (ca la <code>grep</code>)</td></tr>
<tr><td><code>*</code></td><td>Operatorul star: R* = R de zero sau mai multe ori la rînd</td></tr>
<tr><td><code>+</code></td><td>R+ = RR* (R cel puţin o dată)</td></tr>
<tr><td><code>?</code></td><td>R+ = $\epsilon|R$ (R de zero sau una ori)</td></tr>
<tr><td><code>{n}</code></td><td>R{n} = R exact de n ori</td></tr>
<tr><td><code>{n,}</code></td><td>R{n,} = R de cel puţin n ori</td></tr>
<tr><td><code>{n,m}</code></td><td>R{n,m} = R de cel puţin n dar cel mult m ori la rînd</td></tr>
<tr><td><code>\t</code></td><td>Caracterul tab</td></tr>
<tr><td><code>\n</code></td><td>Caracterul sfîrşit de linie</td></tr>
<tr><td><code>\r</code></td><td>Caracterul retur de car</td></tr>
<tr><td><code>\x1B</code></td><td>Caracter descris în hexazecimal</td></tr>
<tr><td><code>\l</code></td><td>Următorul caracter micşorat (majuscule devin minuscule)</td></tr>
<tr><td><code>\u</code></td><td>Următorul caracter majorat (minusculele devin majuscule)</td></tr>
<tr><td><code>\w</code></td><td>Orice caracter care formează “cuvinte” (alfanumerice şi _)</td></tr>
<tr><td><code>\W</code></td><td>Orice caracter care nu formează cuvinte</td></tr>
<tr><td><code>\s</code></td><td>Orice spaţiu</td></tr>
<tr><td><code>\S</code></td><td>Orice caracter care nu e spaţiu</td></tr>
<tr><td><code>\d</code></td><td>Orice cifră</td></tr>
<tr><td><code>\D</code></td><td>Orice caracter care nu e cifră</td></tr>
<tr><td><code>\b</code></td><td>O poziţie dintre un cuvînt şi un non-cuvînt (zero caractere)</td></tr>
<tr><td><code>\B</code></td><td>O poziţie care nu este între un cuvînt şi un non-cuvînt</td></tr>
</table>

6 Alte surse de informaţie

Există o grămadă de informaţie despre aceste programe. Pentru utilitarele descrise (`grep`, `sh`, `sed`, `awk`), paginile de manual interactiv (`man`) au o grămadă de informaţii. De asemenea, implementările “free” din proiectul GNU (menţionat adesea în PC Report, vedeţi de pildă articolul meu din iunie 1998) ale acestor utilitare vin cu documentaţii excelente în formă electronică.

Limbajul Perl are o mulţime de informaţii interesante on-line, la `www.perl.com`; toate manualele de bază sunt acolo în formă electronică.

Dacă vreţi să învăţaţi mai mult despre expresii regulate, există o grămadă de cărţi utile. De exemplu “Cartea Dragon”, a lui Aho, Sethi şi Ullman “Compilers : Principles, Techniques, and Tools”, de la Addison-Wesley din 1985, care este o carte clasică despre teoria şi construcţia calculatoarelor. Un tratament foarte teoretic şi elegant puteţi găsi în cartea lui Dexter Kozen “Automata and Computability”, de la Springer Verlag, 1999.

Echivalentul lui `lex` pentru Java are o pagina de web la `http://www.jflex.de/`, cu tot cu documentaţie.

7 Concluzii

Textul asta prezintă o grămadă de detalii; mă şi întreb dacă vor fi folositoare cuiva... Cred ca unele dintre lucrurile de aici sunt însă demne de reţinut:

- O expresie regulată este o metodă de a descrie o mulţime potenţial infinită de cuvinte printr-un singur şir de caractere;
- Expresiile regulate sunt extrem de folosite în prelucrarea de texte, scrierea de programe, generarea de pagini de web, şi multe alte domenii;
- Există metode practice extrem de eficiente de a genera din expresiile regulate automate mici şi rapide care pot verifica în timp linear dacă un cuvînt dat este generat de expresia regulată;
- N-am spus-o explicit, dar teoria automatelor finite are enorm de multe aplicaţii în afara limbajelor, mai ales în hardware. Dacă veţi citi despre arhitectura sistemelor de calcul veţi vedea că practic *toate microprocesoarele moderne sunt automate finite.*

Orice programator poate deci folosi expresiile regulate, fără să ştie tot ce se ascunde în spatele lor. Nici articolul acesta nu face decît să ridice puţin voalul, făcînd aluzie la o combinaţie de solide rezultate teoretice şi inginereşti. Fundamentele solide sunt cele care de fapt fac această “tehnologie” atît de eficace.