Un PARSER de expresii REGULATE

Alexandru Șușu

Programul GRAPE, despre care vom vorbi în următoarele pagini, este un parser de expresii regulate și a fost implementat de către autor împreună cu un colaborator (Daniel Cuclea, student la Universitatea Politehnica din București), folosind limbajul C++. Sursele acestui program pot fi descărcate de la adresa www.ginfo.ro/revista/11_8/.

Pentru început, trebuie să avertizăm cititorii că articolul conține o mulțime de cuvinte preluate din limba engleză, de care am considerat că este mai bine să nu ne dispensăm, pentru a vă familiariza cu termenii de specialitate. Un alt motiv la fel de important îl constituie inexistența unor traduceri acceptabile în limba română pentru unii termeni.

De ce expresii regulate?

În jurul anilor 1960, perioadă în care se puneau bazele primelor compilatoare, cercetătorii au observat că este mai simplu de realizat și de portat un compilator dacă se proiectează în mai multe blocuri, fiecare bloc fiind specializat pe un anumit proces. Primul bloc al unui compilator poartă numele de ***analizor lexical*** și are următoarele atribuții:

- recunoaște și verifică, după o serie de criterii, validitatea "cuvintelor" (entități care se numesc ***token***-uri) din codul sursă al programului scris în respectivul limbaj de programare (cuvintele cheie, identificatorii și constantele numerice etc.) și le trimite către următorul bloc din compilator;
- dacă analizorul lexical întâlnește un *token* invalid (de exemplu, în cazul limbajelor *Pascal* sau *C/C++*, un identificator de funcție care începe cu o cifră), atunci apelează o rutină de tratare a erorii. Trebuie remarcat faptul că, dacă blocul de analiză lexicală nu găsește nici o eroare, aceasta nu înseamnă că programul este corect. În această fază se aplică doar o parte a criteriilor de validitate pe care un program trebuie să le respecte.

S-a dorit să se proiecteze un analizor lexical într-un mod cât mai elegant. Pentru aceasta s-au pus bazele unui aparat teoretic care să ajute la recunoașterea "cuvintelor" programelor. Poate vă veți întreba care este motivul unor astfel de complicații, deoarece s-ar putea folosi bibliotecile și funcțiile pe care le avem deja la dispoziție. În cazul în care ar trebui să recunoaștem cuvinte ca "for", "while", este adevărat că ar fi suficientă apelarea unei funcții de tipul `strcmp()`.

Dificultatea apare în situația în care trebuie să recunoaștem un identificator (de exemplu, o funcție definită de utilizator), deoarece numărul acestor identificatori poate fi foarte mare. Pentru a accepta identificatorul ca fiind valid trebuie să verificăm dacă primul caracter este literă (sau caracterul `'_'`) și dacă restul caracterelor sunt alfanumerice (sau alte caractere speciale admise, cum ar fi `'_'`, `'$'`). Totuși, putem implementa rutine destul de simple care recunosc șirurile de caractere care sunt identificatori de funcții.

Lucrurile încep să se complice mai mult în cazul constantelor numerice: constantele reprezentate în virgulă mobilă pot fi scrise sub formă științifică cu exponent (de exemplu, `1.894E-3`). Recunoașterea constantelor numerice presupune o rutină în care se verifică destul de multe condiții: dacă există caracterul `'.'`, atunci el trebuie să fie urmat de cel puțin o cifră; dacă există caracterul `'E'` (sau `'e'`) acesta trebuie să fie urmat de un număr întreg etc.

Fiecare dintre cazurile de mai sus au câteva trăsături comune. Astfel, în loc să se trateze separat fiecare caz, se poate concepe o bibliotecă de funcții de recunoaștere mai puternică, dar și mai complexă, care să permită tratarea fiecărui caz prin particularizare. Se poate aduce următorul

contraargument acestei soluții: *o implementare mai generală este cu siguranță mai puțin eficientă decât una care se folosește de toate particularitățile problemei*. Fără îndoială, afirmația este adevărată dar, în cazul programelor de o complexitate ridicată, se recomandă folosirea unei soluții generale (care promite un ordin de complexitate *similar-polinomial* cu alte soluții particularizate), având în vedere riscul mai mic de apariție a unor erori de programare.

Pentru tratarea unificată a analizei lexicale s-a dezvoltat ***teoria expresiilor regulate***, la ora actuală existând în practică o suită impresionantă de aplicații care folosesc din plin acest tip de expresii.

Ce este o expresie regulată?

Expresiile regulate se folosesc pentru a căuta șiruri de caractere specificate prin informații incomplete privind modelul care trebuie găsit. Ele descriu parțial șirurile de caractere pe care dorim să le căutăm. În cazul în care un șir corespunde descrierii date de expresia regulată spunem că s-a găsit o ***potrivire a expresiei regulate peste șirul respectiv***. Cel mai simplu exemplu este cel al folosirii comenzilor *DOS*, de tipul `"dir *.txt"` sau `"dir a?b.txt"`. *Wildcard*-urile folosite în descrierea numelor fișierelor sunt un exemplu nestandard de expresii regulate.

Expresia `"*.txt"` descrie incomplet o clasă largă de șiruri de caractere, anume cele care au ca sufix șirul `".txt"`, adică cele care au extensia `"txt"`. Un nume de fișier care corespunde acestei expresii este `"setup.txt"`. Expresia `"*.txt"` nu se potrivește peste numele de fișier `"io.sys"` (pentru că extensia este `"sys"`).

Expresia `"a?c.txt"` descrie incomplet o altă clasă de șiruri de caractere, și anume cele care au primul caracter `'a'`, urmat de orice caracter, urmat de șirul `"b.txt"`. Așadar, `"abc.txt"` și `"acc.txt"` corespund expresiei, dar șirul `"acb.txt"` nu.

În concluzie, o expresie regulată poate descrie o mulțime foarte mare de șiruri de caractere. Această mulțime se numește ***limbajul*** respectivei expresii. Ne punem acum problema cum se scriu corect aceste expresii regulate. Trebuie avut în vedere următorul aspect: mulțimea tuturor expresiilor regulate (valide, bineînțeles) alcătuiesc limbajul expresiilor regulate; însă la rândul lor, fiecare expresie descrie un limbaj.

Am dori să vă prezentăm fundamentul mecanismului de rulare al programului `GRAPE`: fie o expresie regulată și un text; `GRAPE` va verifica validitatea expresiei (dacă nu este validă va returna un mesaj de eroare) iar, în cazul în care expresia este validă, va returna cel mai lung subșir de caractere din text care se potrivește peste expresia respectivă.

Există mai multe variante de limbaje de expresii regulate. Probabil cel mai simplu dintre acestea este sistemul de specificare a fișierelor cu *wildcard*-uri din *DOS* (sau din *shell*-ul *Unix*). Un limbaj mult mai complex, cu o putere mai mare de exprimare este cel descris în standardul `POSIX 1003.2` (***POSIX*** - ***P**ortable **O**perating **S**ystem un**IX*** - este o specificație dezvoltată pentru standardizarea interfețelor sistemelor de operare de tip *UNIX*), care descrie două limbaje de expresii regulate: unul "*basic*" (care este vechi, dar este păstrat din motive de compatibilitate cu aplicațiile mai vechi) și unul extins.

tabelul 1

Operator	Tip	Simbol	Descriere
concatenare	unar	*juxtapunere*	dacă doi operanzi sunt adiacenți în expresie, atunci avem o potrivire dacă și numai dacă două subșiruri de caractere corespunzătoare celor doi operanzi sunt adiacente în text, în ordinea corespunzătoare; de exemplu, expresia `"AB"` se potrivește peste șirul de caractere `"AB"` și nu se potrivește peste nici un alt șir de caractere
sau	binar	`'|'`	ne permite să specificăm mai multe alternative care pot fi luate în considerare în cazul căutării unei potriviri; expresia `"A|B"` se potrivește doar peste două șiruri de caractere: `"A"` și `"B"`; expresia `"C(A|U)(I|L)"` se potrivește doar peste următoarele patru șiruri de caractere: `"CAI"`, `"CAL"`, `"CUI"` și `"CUL"`
Kleen-closure	unar	`'*'`	aplicat unui operand, indică faptul că respectivul operand poate apărea concatenat cu el însuși de oricâte ori (numărul aparițiilor poate fi și zero); expresia `"A*"` se potrivește peste orice șir de caractere care conține doar caractere `'A'` (inclusiv șirul vid); expresia `"(A|B)*"` se potrivește cu orice șir care conține doar caractere `'A'` și `'B'` (inclusiv șirul vid), de exemplu `"A"`, `"B"`, `"ABBA"`, `"BABA"` etc
paranteze	unar	`'('`, `')'`	prin gruparea simbolurilor este posibilă schimbarea priorității operatorilor
orice caracter	unar	`'.'`	expresia `"A."` se potrivește peste `"AB"`, `"AC"` etc; se observă că acest operand este echivalent cu expresia formată din disjuncția tuturor celorlalți operanzi prezentați, deci acesta nu este un operand elementar

Operanzii sublimbajului expresiilor regulate care este tratat în GRAPE

O expresie regulată este foarte asemănătoare din punct de vedere constitutiv cu o expresie aritmetică, ambele fiind compuse din operatori și operanzi.

În programul GRAPE am tratat un *sublimbaj* al limbajului expresiilor regulate extinse. Acest sublimbaj conține cei mai importanți operatori și operanzi. Operatorii sunt prezentați în tabelul 1.

Am putea să încheiem aici descrierea limbajului, dar îl vom îmbogați cu încă un element util. Să presupunem că dorim să căutăm într-un text o linie care conține caractere `'*'`. Trebuie să găsim o metodă pentru a face distincție între operandul (caracterul) `'*'` și operatorul *Kleene-closure* `'*'`. O soluție elegantă o reprezintă folosirea unui "***escape char***" - `'\'` care anulează semnificația specială a caracterului care urmează. Introducem așadar în limbaj mai mulți operanzi formați din perechea de caractere `'\'` și unul din caracterele speciale (operatorii) prezentați anterior, care transformă caracterul special în caracter obișnuit care va trebuie potrivit. Apare întrebarea: *cum se poate reprezenta operandul care se potrivește cu caracterul* `'\'`? Răspunsul este simplu: *îl vom prefixa cu un* ***escape char***. Reprezentarea acestui operand va fi `"\\"`.

Așadar, operanzii și operatorii din care sunt formate expresiile regulate sunt:

Simbol	Semnificație
`'A'-'Z','a'-'z'`	*litere*
`'0'-'9'`	*cifre*
`'.'`	*orice caracter*
`'(',')'`	*paranteze*
`'|'`	*disjuncție*
`'*'`	*închidere Kleene*
`"\(","\)","\*","\.","\\"`	*modificator*

Atribuirea acestei semnificații caracterului `'\'` permite reprezentarea caracterelor `'('`, `')'`, `'*'` și `'.'` fără semnificația lor specială, astfel încât să poată fi folosiți ca simpli operanzi (caractere care trebuie potrivite peste text). Pentru a putea folosi și caracterul `'\'` pe post de operand a fost introdus și operatorul `"\\"`.

În standardul POSIX 1003.2 mai există o serie de operanzi și operatori utili, care nu sunt implementați în programul GRAPE. Câțiva dintre aceștia sunt prezentați în tabelul 2.

Simplul fapt că știm care sunt caracterele care pot compune o expresie regulată nu înseamnă că am descris complet limbajul expresiilor regulate. Trebuie să stabilim niște reguli de interacțiune a acestora.

Pentru specificarea exactă a sintaxei limbajului expresiilor regulate se folosește un obiect matematic denumit ***gramatică independentă de context***.

În [5] se specifică faptul că o gramatică este un set de reguli privitoare, pe de o parte, la forma și modificarea cuvintelor (***morfologia***), iar, pe de altă parte, la îmbinarea cuvintelor în procesul comunicării (***sintaxa***). Tot în [5] se precizează că gramaticile se pot împărți în gramatici destinate marelui public și în gramatici adresate specialiștilor, acestea din urmă putând fi tradiționale sau moderne. Cele moderne pot fi de mai multe tipuri, printre care există și categoria gramaticilor matematice care ne interesează.

Gramaticile independente de context au fost introduse în 1956 de către ***Noam Chomsky***, în timp ce studia limbajele naturale. Adoptarea gramaticilor independente de context în cadrul științei calculatoarelor a fost realizată de către ***John Backus*** în timp ce lucra la o schiță a raportului limbajului de programare *Algol 60* (un "bunic" al limbajului *C*), pentru a descrie sintaxa respectivului limbaj.

Pentru aceasta s-a folosit o notație denumită la început *Backus-Normal Form* (***BNF***) (în 1964 ***Donald Knuth*** a propus într-o scrisoare ca notația să se numească ***Backus-Naur Form***, în onoarea lui ***Naur***, alt cercetător care a participat la redactarea raportului *Algol 60*). Notația *BNF* a gramaticii limbajului expresiilor regulate este:

```
<expression> ::= <term> |
                 <term>OR_OPERATOR<expression>
<term> ::= <factor> | <factor><term>
<factor>::=(<expression>) | ch |
              <factor>KLEENE_CLOSURE_OPERATOR,
```

unde `ch` reprezintă oricare dintre operanzi.

tabelul 2

Operator	Tip	Simbol	Descriere
listă de caractere	operand	`[N-M]`	permite o scriere mai compactă a expresiilor; de exemplu, expresia `"[0-9]"` este echivalentă cu expresia `"(0|1|2|3|4|5| 6|7|8|9)"`
complementare	operator	`'^'`	un exemplu în care se folosește acest operator este expresia `"[^0-9]"` care se potrivește peste toate șirurile de un caracter cu excepția celor care conțin o cifră
expresie interval	operator	`{N,M}`	expresia asupra căreia se aplică se poate repeta de cel puțin *N* ori și de cel mult *M* ori; de exemplu, expresia `"A{1,3}"` este echivalentă cu `"A|(AA)|AAA"`
semn de întrebare	operator	`'?'`	indică cel mult o apariție a operandului căruia i se aplică
back-referencing	operator	`'\n'`	potrivește peste subșirul potrivit anterior de a `n`-a subexpresie parantezată din expresia regulată. De exemplu, expresia `"(A*)B\1"` se potrivește peste șirul `"AAABAAA"`, dar nu se potrivește peste șirul `"AAABAAB"`

Operatori și operanzi specificați de standardul POSIX 1003.2

Am folosit notațiile KLEENE_CLOSURE_OPERATOR și OR_OPERATOR pentru operatorii ***Kleene-Closure*** și ***sau***, din dorința de a elimina ambiguitățile cu privire la semnificația caracterului '|'.

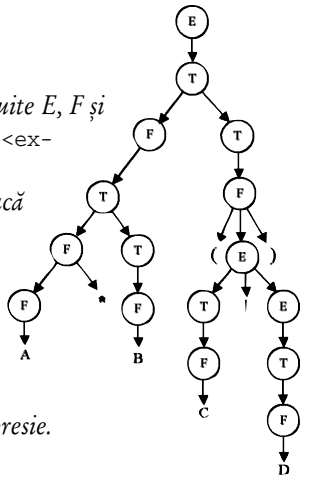
Această notație conține următoarele elemente:

- ***producții***, care sunt compuse din ***simboluri terminale*** (simboluri care aparțin limbajului: ch, OR_OPERATOR, KLEENE_CLOSURE_OPERATOR și parantezele rotunde), ***simboluri neterminale*** (simboluri interne gramaticii: <expression>, <term>, <factor>) și ***metasimboluri*** ("::=" și "|"). Metasimbolul "::=" are semnificația ***"este"***; el indică faptul că membrul stâng are, prin definiție, aceeași semnificație ca și membrul drept. Metasimbolul "|" reprezintă disjuncția celor doi termeni din stânga, respectiv din dreapta sa.
- neterminalul din membrul stâng din prima producție (<expression>) se numește ***simbol de start*** al gramaticii;

Cea de-a doua producție indică faptul că <term> reprezintă un <factor> sau un <factor> concatenat cu <term>. Observăm ca avem o definiție recursivă pe care, prin raționament inductiv, o putem extinde la următoarea definiție: <term> este o succesiune nevidă (cel puțin un element) de <factor>-i.

O metodă naturală de a descrie un proces format din mai multe derivări este ***arborele de derivare***. Arborii de derivare sunt foarte intuitivi, motiv pentru care vom da direct un exemplu de arbore.

Considerăm expresia A*B(C|D), al cărei arbore de derivare (***parse tree***) este prezentat în figură.



Nodurile interne încercuite E, F și T reprezintă neterminalii <expression>, <factor> *și* <term>. *Săgețile care pleacă de la un neterminal se îndreaptă spre simbolurile în care derivează respectivul neterminal.*

Arborele descrie toate etapele analizei sintactice având data respectiva expresie.

Din notația *BNF* a gramaticii se pot deduce următoarele proprietăți ale operatorilor:

- OR_OPERATOR are prioritatea mai mică decât operatorul concatenare, iar operatorul concatenare are prioritatea mai mică decât KLEENE_CLOSURE_OPERATOR.
- OR_OPERATOR și operatorul ***concatenare*** sunt asociativi la dreapta.

Exemple

Vom prezenta acum două exemple concrete de folosire a expresiilor regulate:

- clasa șirurilor de caractere care descriu numere reale de tipul 01, 12, 12.3, 12.3E4 poate fi descrisă folosind următoarea expresie regulată:

```
"[0-9]+(.[0-9]+)?(E(+|-)?digit+)?"
```

- expresia regulată care ar descrie numele valide de funcții în *C* este:

```
"(_|$|[a-zA-Z]) (_|$|[a-zA-Z0-9])*".
```

Cum se potrivește un șir peste o expresie regulată?

Pentru a rezolva sarcina potrivirii unui șir de caractere cu o expresie regulată, trebuie ca aceasta să fie transformată într-o formă mai accesibilă calculatorului. Teoria ne spune că pentru orice expresie regulată există cel puțin un ***A**utomat **F**init **N**edeterminist* (***AFN***) și cel puțin un ***A**utomat **F**init **D**eterminist* (***AFD***) care sunt echivalente cu expresia [2].

Programul GRAPE realizează conversia unei expresii regulate într-un *AFN* pe care îl folosește pentru a accepta șirurile de caractere care se potrivesc peste respectiva expresie.

Ce este un AFN?

În primul rând, un automat finit este un obiect matematic care are o mulțime finită de stări, evoluând de la o stare la alta pe baza unor date (caractere) de intrare.

Un *AFN* (întâlnit în literatura de specialitate și sub numele de ***syntax charts*** sau ***railroad normal form***) este un graf orientat în care nodurile reprezintă stările pe care le poate avea automatul la momente de timp distincte.

AFN-ul are desemnate o stare inițială (de *start*) și mai multe stări finale (*acceptoare*). Muchiile dintre stări se numesc tranziții, fiecare tranziție având asociat un caracter care indică faptul că se poate ajunge din starea curentă într-o altă stare, pe tranziția respectivă, doar atunci când caracterul din textul de intrare coincide cu caracterul tranziției. Printre caracterele care se pot asocia unei tranziții se introduce și caracterul nul - ε, prezența sa indicând faptul că tranziția se poate executa indiferent care este caracterul de la intrare.

Caracterul nedeterminist al *AFN*-ului este ilustrat de posibilitatea ca dintr-o stare să poată exista mai multe stări la care să se poată ajunge pe baza unui același caracter. Exact aceasta este diferența dintre un *AFN* și un *AFD*: un *AFD* are cel mult o stare în care se poate ajunge din starea curentă pe baza caracterului de intrare, pe când un *AFN* poate avea o mulțime de astfel de stări.

AFN-ul implementat în GRAPE acceptă șiruri de caractere din mulțimea: {'a'-'z', 'A'-'Z', '0'-'9', '.', '*', '\', '|', '(', ')'}. Mulțimea de caractere care se pot asocia unei tranziții este cea descrisă anterior reunită cu {ε}. În cazul unui șir de caractere de intrare pentru care dorim să determinăm dacă este acceptat de *AFN*, va trebui să ***executăm*** *AFN*-ul; această operație se realizează astfel:

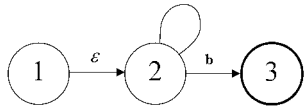
- inițial, *AFN*-ul se află în starea de start;
- când *AFN*-ul primește un caracter din textul de intrare, atunci el evoluează în funcție de caracterul primit astfel:

- dacă nu există tranziție care să iasă din starea curentă și care să conțină caracterul de intrare curent, atunci *AFN*-ul anunță exteriorul că nu s-a găsit potrivire între șirul de caractere de intrare și expresia regulată, și revine în starea inițială, așteptând alte secvențe de caractere;
- dacă există o tranziție din starea curentă pe baza caracterului de intrare curent, atunci *AFN*-ul execută tranziția respectivă; dacă există mai multe tranziții din starea curentă pe baza aceluiași caracter de intrare, atunci *AFN*-ul are puterea teoretică de a ghici tranziția care duce spre starea finală (spunem că *AFN*-ul este înzestrat cu puterea nedeterminismului, acest lucru înseamnă că reușește, având de ales dintre mai multe decizii, să o aleagă pe cea corectă); în continuare vom vedea cum se poate implementa o simulare de execuție a *AFN*-ului pe un calculator numeric, cu toate că pare paradoxal să se poată simula un automat nedeterminist, pe o mașină deterministă.

• când se ajunge într-una din stările finale, *AFN*-ul se resetează, anunță exteriorul că s-a găsit o potrivire și revine în starea inițială, așteptând alte secvențe de caractere.

Exemplu

Vom considera automatul finit nedeterminist din figura alăturată. Starea inițială a *AFN*-ului este 1. Cercul îngroșat care simbolizează starea 3 indică faptul că aceasta este o ***stare acceptoare***.



Presupunem că avem șirul de intrare `"ab"`. Inițial *AFN*-ul este în starea 1, iar caracterul curent de la intrare este primul caracter al șirului. Se execută tranziția (fără a se citi caracterul curent de la intrare) și se ajunge astfel în starea 2. Pentru caracterul de intrare `'a'`, executăm tranziția cu `'a'`, ajungând tot în starea 2. Caracterul curent de intrare devine cel de-al doilea caracter, `'b'`. Se execută tranziția cu `'b'` și se ajunge în starea finală 3, deci șirul `"ab"` este acceptat de *AFN*.

Analog se poate vedea că șirurile `"b"`, `"aab"`, `"aaab"` sunt acceptate de *AFN*, în timp ce șirurile `"a"`, `"ac"`, `"acb"` nu sunt acceptate. De fapt se observă că limbajul șirurilor acceptate de *AFN* este exact limbajul descris de expresia regulată `"a*b"`.

Cum se construiește un AFN dintr-o expresie regulată?

O expresie regulată poate avea mai multe *AFN*-uri echivalente. *AFN*-urile respective pot avea un număr diferit de stări și structuri distincte, dar execuțiile lor trebuie să fie echivalente. Există un algoritm (algoritmul de construcție al lui ***Thompson***) care, pe baza arborelui de derivare generat din expresia regulată, poate construi un *AFN* cu următoarele proprietăți adiționale:

• *AFN*-ul are o singură stare finală;

• fiecare stare a *AFN*-ului are:

- fie o singură tranziție non-ε și nimic altceva;
- fie cel mult două tranziții ε și nimic altceva, caz în care se numește ***null-state***.

Pentru a transforma (mai științific, ***a compila***) o expresie regulată într-un *AFN* trebuie să apelăm, după cum am spus, la arborele de derivare. Pentru aceasta trebuie realizată analiza sintactică a expresiei regulate. Blocul care realizează această operație este denumit în teoria compilatoarelor ***analizor sintactic*** și este blocul care urmează după analizorul lexical. Funcția analizorului sintactic (a *parser*-ului) este de a determina dacă șirul de caractere de la intrare este o expresie regulată și de a construi arborele de derivare corespunzător.

Algoritmul lui Thompson...

...pentru construcția unui AFN dintr-o expresie regulată

Intrare

Se consideră o expresie regulată *r* peste alfabetul de caractere acceptate Σ.

Ieșire

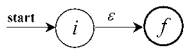
AFN-ul corespunzător expresiei.

Metoda

Mai întâi se va parsa expresia în subexpresiile sale, apoi utilizând regulile (1) și (2), pe care le vom prezenta în continuare, se vor construi *AFN*-uri pentru fiecare simbol de bază (acesta poate fie un simbol din alfabet, fie ε). Este important de observat că, dacă un simbol apare de mai multe ori în *r*, atunci va fi construit câte un *AFN* pentru fiecare apariție a sa.

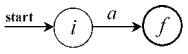
Apoi, în funcție de structura sintactică a expresiei regulate se vor combina aceste *AFN*-uri, folosind regula (3), până când se va obține *AFN*-ul pentru întreaga expresie.

(1). Pentru ε construim *AFN*-ul:



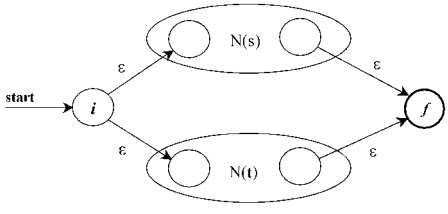
unde *i* este starea inițială, iar *f* este starea finală.

(2) Pentru *a* din Σ construim *AFN*-ul:

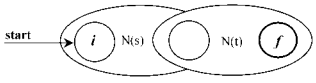


(3) Presupunem că *N*(*s*) și *N*(*t*) sunt *AFN*-urile pentru expresiile *s* și *t*.

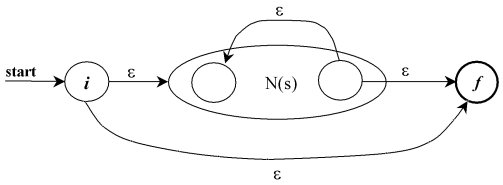
a) Pentru expresia regulată `s|t`, se va construi următorul *AFN*:



b) Pentru expresia regulată `st`, se va construi *AFN*-ul:



c) Pentru expresia regulată `s*` vom avea:



d) În cazul în care expresia regulată este în paranteză `(s)` vom construi *AFN*-ul pentru *N*(*s*) și vom ignora parantezele.

Se observă că algoritmul introduce cel mult două stări în *AFN* pentru fiecare caracter din expresia regulată. Așadar, putem afirma că *AFN*-ul are cel mult de două ori mai multe stări decât numărul de caractere din care este construită expresia regulată.

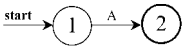
Exemplu

Vom construi *AFN*-ul pentru expresia regulată

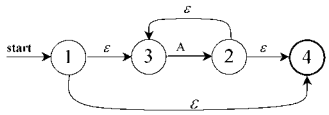
```
r = A*B(C|D),
```

al cărei arbore de derivare a fost prezentat anterior.

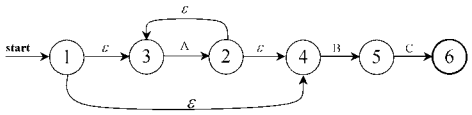
Pentru subexpresia `"A"` (primul caracter din expresie), aplicând regula (2) obținem *AFN*-ul:



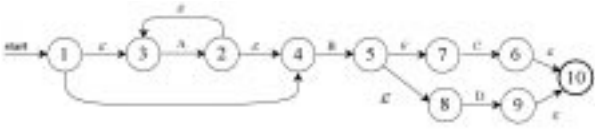
La pasul următor întâlnim operatorul `'*'`, aplicăm regula (3.c) și obținem:



Aplicăm regula (2) când întâlnim subexpresia `"B"`, iar când întâlnim subexpresia `"(C|D)"`, memorăm faptul că starea 5 este starea de început a acestei subexpresii și, inițial, prelucrăm doar subexpresia `"C"`:



Știm că starea 5 este starea de început a subexpresiei `"(C|D)"`. Când întâlnim operatorul `'|'`, aplicăm regula (3.a) și obținem forma finală a *AFN*-ului:



Se impun câteva explicații:

- etapele creării *AFN*-ului prezentate mai sus sunt exact etapele pe care le parcurge și programul `GRAPE` când convertește o expresie regulată în *AFN*.
- s-ar putea să vă mirați de faptul că o stare creată la un anumit moment își poate schimba numărul de ordine la un pas ulterior. Așa este concepută implementarea algoritmului lui *Thompson* în `GRAPE`. Puteți studia sursa programului `GRAPE`, ea fiind cea mai grăitoare descriere a implementării.

Prezentarea analizorului sintactic

Am spus că algoritmul de construcție a *AFN*-ului din expresia regulată apelează la *parser* (analizor sintactic). Metoda de parsare care este folosită de programul `GRAPE` este denumită ***top-down parsing*** (sau ***recursive-descent***). Ea funcționează în felul următor: pentru o expresie, arborele de derivare corespunzător se construiește pornind de la rădăcină până când se ajunge la frunze (de aici provine denumirea *top-down*).

Metoda se implementează utilizând descrierea *BNF* a gramaticii expresiilor regulate; mai exact, realizăm o "traducere" în limbaj de programare a fiecărei producții a gramaticii. Așadar, analizorul este compus din următoarele funcții: `Expression`, `Factor` și `Term`.

Aceste funcții conțin, pe lângă "traducerile" producțiilor corespunzătoare, și acțiunile de construire a *AFN*-ului (științific, aceste acțiuni sunt numite ***acțiuni semantice***).

Cum este reprezentat intern un AFN?

Proprietățile *AFN*-ului construit cu ajutorul algoritmului lui *Thompson* ne determină să folosim următoarea reprezentare pentru acesta:

- un vector `ch`, unde elementul arbitrar `ch[i]` reprezintă caracterul asociat tranziției care pornește din starea numerotată cu *i*. Acest caracter poate să fie unul acceptat de *AFN*, sau poate fi `NULL_STATE_CHAR` (ceea ce semnifică faptul că starea respectivă este *null-state*);
- doi vectori `next1` și `next2`, unde `next1[i]` și `next2[i]` reprezintă stările în care sosesc tranzițiile care pornesc din starea *i*. Vectorii pot avea valori `INVALID_TRANSITION`, ceea ce înseamnă că nu există tranziția respectivă.
- cei trei vectori sunt statici și au aceeași dimensiune: dublul lungimii maxime acceptate a șirului expresiei regulate.

Cum se execută AFN-ul?

După construirea *AFN*-ului, putem să verificăm dacă respectivul automat acceptă anumite șiruri de caractere. Aceasta se realizează cu funcția:

```
int Match(char *text),
```

unde `text` este șirul de caractere *ASCII* care va fi sau nu acceptat de către *AFN*, iar valoarea returnată de funcție reprezintă numărul de caractere conținute de subșirul prefix de lungime maximă din text, care se potrivește peste expresia regulată echivalentă cu *AFN*-ul (dacă rezultatul este un întreg strict pozitiv).

Implementarea acestei funcții folosește structura de date ***DEQue*** (***D**ouble-**E**nded **Que**ue*), o structură la care se pot adăuga elemente în capul (funcția `AddHead()`) și coada structurii (funcția `AddTail()`), dar din care elementele se extrag doar de la cap (funcția `GetHead()`). Această structură combină proprietățile unei cozi cu cele ale unei stive.

În *DEQue* există, în orice moment, două tipuri de stări: unele sunt depozitate în "jumătatea" superioară a structurii (care conține capul structurii), iar celelalte în "jumătatea" inferioară.

Pentru a se delimita cele două "jumătăți" (care pot fi inegale) se folosește un element special, denumit `SCAN_MARKER`, care nu poate apărea în nici una dintre cele două "jumătăți".

În "jumătatea" superioară se memorează stările în care se poate afla *AFN*-ul înainte de a potrivi caracterul curent din șirul de intrare, iar în "jumătatea" inferioară stările fii ale stărilor din prima "jumătate" care au potrivit caracterul curent din șirul de intrare.

Observație
Execuția *AFN*-ului este un caz particular de parcurgere în lățime.

Exemplu
Vom descrie acum modalitatea de funcționare a funcției `Match()`, exemplificând pe *AFN*-ul construit la exemplul anterior și șirul de intrare `AABDE`.

Inițial, trebuie potrivit primul caracter din text, iar în *DEQue* avem:

1
SCAN_MARKER

Extragem `1` din coadă. Observăm că din starea `1` se poate ajunge în stările `3` și `4` prin tranziții ε, așadar *DEQue* va arăta astfel:

4
3
SCAN_MARKER

Extragem `4` din coadă: starea `4` are o singură tranziție cu caracter `'B'` asociat, deci nu putem să folosim starea `4`. Extragem starea `3` din coadă: starea `3` are o tranziție cu caracter `'A'` asociat, deci executăm `AddTail()` cu starea fiu a stării `3`. *DEQue* va fi:

SCAN_MARKER
2

Ajungem să extragem din coadă elementul `SCAN_MARKER`; vom înainta la următorul caracter din text și vom insera un element `SCAN_MARKER` în coadă:

2
SCAN_MARKER

Din starea `2` se poate ajunge în stările `3` și `4` prin tranziții ε, așadar *DEQue* va fi:

4
3
SCAN_MARKER

Din nou potrivim cu tranziția spre starea `2` cel de-al doilea caracter `'A'` din text.

SCAN_MARKER
2

Întâlnim `SCAN_MARKER`; înaintăm la al treilea caracter din text, `'B'`. *DEQue* va fi:

2
SCAN_MARKER

După aceea *DEQue* va fi din nou:

4
3
SCAN_MARKER

Caracterul curent din text este potrivit doar de starea `4`, așa că înaintăm la caracterul `'D'` din text. *DEQue* va fi:

5
SCAN_MARKER

Din starea `5` se poate ajunge prin tranziții ε la stările `7` și `8`:

8
7
SCAN_MARKER

Caracterul curent din text este potrivit doar de starea `8`; înaintăm la caracterul `'E'` din text. *DEQue* va fi:

9
SCAN_MARKER

Din starea `9` se ajunge prin ε-tranziție în starea `10`. *DEQue* va fi:

10
SCAN_MARKER

În momentul în care extragem starea `10` din coadă, observăm că `10` este stare finală. Așadar șirul `"AABD"` este acceptat de *AFN*. Drept urmare, funcția va returna valoarea `4` (lungimea maximă a șirului prefix al textului de intrare care este acceptat de *AFN*).

Observație
AFN-ul construit prin algoritmul *Thompson* poate conține cicluri compuse doar din *null-state*-uri. În acest caz, este posibil ca, atunci când calculăm stările în care se poate ajunge prin ε-tranziții dintr-o stare, să intrăm într-o buclă infinită, care să ducă la umplerea până la refuz a structurii *DEQue*. Există două soluții pentru această problemă:

- folosirea unui vector de elemente *booleene* care să indice dacă am mai trecut sau nu prin starea la care ajungem prin ε-tranziții din starea curentă. (Aceasta metodă este folosită în implementarea standard a programului `GRAPE`.)
- Mulțimile de stări în care se poate ajunge din orice stare prin ε-tranziții (implementarea *eps_closure*) se pot determina înainte de apelul funcției `Match()`.

Pentru a găsi subșirul (nu neapărat prefix) de lugime maximă, acceptat de *AFN*, se vor executa apeluri succesive de tipul `Match(text)`, `Match(text+1)`, ..., `Match(text+strlen(text)-1)`, memorându-se poziția și lungimea prefixului maximal găsit de toate funcțiile `Match()` apelate.

Analiza complexității programului

Timpul de execuție al *parser*-ului are un ordin de complexitate proporțional cu L, unde L este numărul de caractere al expresiei regulate.

Timpul de execuție al funcției `Match()` are un ordin de complexitate $O(M \cdot N)$, unde M este numărul de stări din care este compus *AFN*-ul, iar N este numărul de caractere ale textului în care se încearcă potrivirea.

Necesarul de memorie are ordinul de complexitate $O(M)$.

Alte utilizări ale expresiilor regulate

Expresiile regulate au fost studiate pentru prima dată de către *Kleene* în anul 1956; el era interesat în a descrie evenimentele reprezentate de automatele finite care modelează activitatea neuronală.

Datorită faptului că sunt foarte descriptive:

- se pot folosi în găsirea cuvintelor scrise greșit; de exemplu expresia regulată `".*(ie+ei).*"` găsește cuvintele care conțin `"ie"` sau `"ei"`; aceste cuvinte au o probabilitate relativ mare de a fi scrise greșit în limba engleză.
- pot determina dacă o adresă *e-mail* este scrisă corect sau nu.
- se folosesc la analiza *script*-urilor server *Apache*.
- se folosesc în interogările *SQL*.

Jarvis (în anul 1976) a folosit expresiile regulate pentru a găsi imperfecțiuni în cablajele imprimate.

O mulțime de programe au inclus suport pentru expresii regulate: *SED*, *AWK*, *PERL*, *PHP*, *VIM*, *LEX*, *FLEX*.

La final...

Fișierele sursă ale pachetului `GRAPE` sunt:

- **`COMDEF.H`** - definiții de tipuri de date;
- **`DEQUE.H`** - clasa `DEQue` (*Double Ended Queue*)
- **`NFA.CPP`** și **`NFA.H`** - clasa `AFN` (*Nondeterministic Finite Automata*) - conține funcțiile de analiză lexicală și de potrivire;
- **`GRAPE.CPP`** - interfața programului cu utilizatorul; se citesc expresiile regulate din fișierul **`REGEXP.IN`**, și textul din fișierul **`TEXT.IN`**; se descrie fiecare *AFN* asociat fiecărei expresii regulate în fișierul **`NFA_DIAG.OUT`**; în fișierul **`REGEXP.OUT`** se scriu detalii despre construcția și execuția fiecărui *AFN* pentru textele de intrare.

Notă

Programul `GRAPE` a fost testat de autori și de redacția *GInfo*. Totuși, nici autorul, nici membrii redacției nu pot fi făcuți responsabili pentru eventuale daune provocate de funcționarea incorectă a programului.

Observație

E interesant de remarcat că `GRAPE` este, de fapt, un compilator, compus din:

- un modul de analiză lexicală de o complexitate foarte redusă (care verifică dacă s-a citit sau nu un caracter alfanumeric, dacă `'\'` este caracterul curent etc.; în cazul apariției caracterului `'\'` se trece la caracterul următor prin efectuarea unei citiri suplimentare);
- un modul de analiză sintactică în cadrul căruia se realizează și traducerea expresiei în *AFN*;
- un modul de *matching* (potrivire) despre care se poate spune că realizează interpretarea *AFN*-ului, având ca date de intrare textul.

Trebuie remarcat faptul că, în programul `GRAPE`, modulul de analiză lexicală este îmbinat cu cel de analiză sintactică.

Alte implementări

- **ftp://ftp.zoo.toronto.edu/pub/bookregexp.{tar|shar}** - implementarea unui *parser* de expresii regulate cu *AFN*.
- ***GREP*** (***G**et **R**egular **E**xpression **P**arser*)
- în numărul din decembrie 2000 al revistei *Dr. Dobbs Journal*, a apărut un articol cu titlul "*How Can I Extend Java's Search Capabilities?*", care conține și sursele *Java* ale unui utilitar de tip *GREP*. Acestea au fost publicate online și sunt disponibile în format electronic pe *site*-ul **www.ddj.com**.

Programul GREP

Merită prezentată o scurtă istorie a programului *GREP*: prima versiune a fost scrisă de ***Ken Thompson*** și era o implementare de parser cu *AFN* (ca și `GRAPE`).

În 1976, ***Al Aho*** a realizat o nouă implementare a acestui program (pe care a denumit-o ***EGREP*** - ***E**xtended **GREP***) și care folosește un automat finit determinist. *EGREP* avea următoarele caracteristici: analiza sintactică dura mai mult decât în cazul primei versiuni de *GREP*, dar partea de execuție a automatului finit era mai rapidă. Ca urmare, ***Aho*** a adus o îmbunătățire ulterioară *EGREP*-ului - o tehnică pe care a numit-o "*cached lazy evaluation*", care surclasează celelalte versiuni de *GREP*. Implementarea *GNU GREP* are surse publice.

Bibliografie

[1] **Robert Sedgewick**, *Algorithms*, Addison-Wesley, 1984 - capitolele *Pattern Matching* și *Parsing*

[2] **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman,** *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1986

[3] **Mihai Budiu**, *Expresii regulate*, PC-Report, Computer Press Agora, martie 2000

[4] **Alexander Pereira Calsavara**, *How Can I Extend Java's Search Capabilities?*, Dr. Dobbs Journal, decembrie 2000

[5] **Mioara Avram,** *Gramatica pentru toți*, Editura Academiei RSR, 1986

Alexandru Șușu este student în anul IV la Facultatea de Automatică și Calculatoare din cadrul Universității Politehnica din București. Poate fi contactat prin e-mail la adresa **asusu@home.ro**.