Limbaje formale și tehnici de compilare

Curs 3: limbaje regulate; expresii regulate: sintaxă, semantică, exemple

Limbaje regulate

- Sunt limbaje ale căror reguli se pot implementa folosind un **automat cu stări finite**
- Corespund gramaticilor de tip 3 din clasificarea lui Chomsky (producțiile sunt de forma a→Ab|A sau a→bA|A)
- Implementarea lor este simplă și viteza analizei sintactice este cea mai mare dintre toate clasele de limbaje
- Deficiența lor majoră este faptul că nu admit construcții recursive (*"nu știu să numere"*) și atunci nu pot fi folosite pentru a testa de exemplu împerecheri de paranteze imbricate: *((a+3)*2-1)/5*
- Sunt folosite pentru definirea expresiilor regulate și în analiza lexicală, deoarece regulile lexicale în general nu sunt recursive (un exemplu de recursivitate ar fi dacă limbajul C ar admite comentarii imbricate: /*../* */..*/)

Definirea limbajelor regulate

- Mulțimea limbajelor regulate peste un alfabet A, se definește astfel:
- Limbajul vid {} (∅) și cel care conține doar șirul vid {ϵ} sunt limbaje regulate
- Pentru $\forall a \in A$, {a} este un limbaj regulat
- Dacă A și B sunt limbaje regulate, atunci $A \cup B$ (reuniune), $A \bullet B$ (concatenare, scris simplificat AB) și A^* (repetiție opțională sau *operatorul/închiderea/stea Kleene*) sunt limbaje regulate
- Nici un alt limbaj peste A nu mai este regulat

Expresii regulate (ER)

- Sunt o notație (un formalism) care implementează limbaje regulate. O ER este o regulă care **recunoaște** (match) o secvență de text
- Există mai multe dialecte de ER (ex: POSIX, Perl/PCRE, …), fiecare cu diverși operatori și reguli
- Unele dialecte au operatori care permit să se scrie ER care depășesc clasa limbajelor regulate (ex: *referințe anterioare:* **(a*)\1** → {ϵ, aa, aaaa, aaaaaa} = $\{a^n a^n | n \geq 0\}$)
- Sunt foarte răspândite în multe de domenii: validarea intrărilor, extragere de date, conversii de formate, căutare/înlocuire, analiza lexicală, etc. Practic aproape toate aplicațiile de prelucrare de text beneficiază de pe urma ER.
- Caracterele care au o semnificație specială (ex: **. ( ) * + ?**) se numesc **metacaractere**

Caracterele din ER

- Literele, cifrele și toate caracterele simple (care nu sunt metacaractere) se recunosc pe ele însele (ex: ion va recunoaște toate secvențele de 3 litere *ion* din text). Dacă nu se specifică altfel, se face diferența între literele mari și mici (ex: Mare este diferit de mare).
- Spațiile sunt semnificative și sunt considerate caractere simple (ex: ion ana recunoaște șirul de 7 caractere dat și este diferit de ionana)
- **.** (**punct**) recunoaște orice caracter (ex: ra. → *ram*, *ras*, *rar, ra#*). În funcție de dialectul de ER și de unii modificatori, punctul poate sau nu să recunoască \n (new line).
- \ (**backslash**) pus în fața unui metacaracter îi anulează semnificația specială și acel caracter va fi recunoscut ca el însuși (ex: \\\. → \.)

Clase de caractere

- **[...] – clasă de caractere** – recunoaște orice caracter (unul singur) care face parte dintre caracterele specificate (ex: [aeiou] recunoaște o vocală).
- În interiorul [] se pot defini intervale de caractere, punând – (cratimă) între capetele intervalului (ex: [a-zA-Z] recunoaște orice literă). Dacă – apare la început sau la sfârșit, atunci se consideră caracter obișnuit (ex: [+-])
- **[^...] – clasă de caractere negată** – Caracterul ^ (căciulă / hat) pe prima poziție într-o clasă de caractere are semnificația de "**orice caracter cu excepția celor din clasă**" (ex: [^0-9] va recunoaște un caracter care nu este cifră)
- În interiorul claselor de caractere aproape toate metacaracterele își pierd semnificația specială, cu excepția – \ ] (ex: [().*] va recunoaște paranteze, punct sau steluță). Metacaracterele rămase se pot transforma în caractere simple folosind \ (ex: [\\\]] recunoaște \ sau]) Unele dialecte de ER nu acceptă nici \ în interiorul [].

Operatori (1)

- **e_1e_2 – secvență** – ambele ER, e_1 și e_2, trebuie să fie îndeplinite în ordine (ex: == recunoaște o succesiune de două semne =)
- **e* – repetiție opțională** – *e* se poate repeta de oricâte ori, sau poate lipsi (ex: [a-z][0-9]* – o literă obligatorie urmată opțional de oricâte cifre)
- **e+ – repetiție obligatorie** – *e* trebuie să apară cel puțin o dată (ex: [0-9]+ – cel puțin o cifră)
- **e? – opțional** – *e* poate sau nu să apară (ex: -?[0-9]+ – cratimă opțională, urmată de cel puțin o cifră)
- **$e_1|e_2$ – alternativă** – una dintre cele două ER trebuie să fie adevărată. Ele se testează în ordine și dacă o ER este adevărată iar ER se termină, următoarele ER nu mai sunt testate (ex: \?|! – testează dacă apare semnul întrebării sau al exclamării)

Operatori (2)

- **e{n} – repetă *e* de *n* ori** (ex: [0-9]{4} o cifră repetată exact de 4 ori)
- **e{n,} – repetă *e* de minim *n* ori** (ex: [a-z]{2,} cuvinte de minim 2 litere)
- **e{m,n} – repetă *e* între *m* și *n* ori** (ex: [0-9]{1,4} numere între 1 și 4 cifre)
- **^e – recunoaște e doar dacă ea este la început de linie sau șir.** ^ în sine nu consumă niciun caracter, ci doar ancorează ER în acea poziție.
  Atenție: dacă ^ este în interiorul [], el are semnificația prezentată anterior la clase de caractere. (ex: ^#[^\n]* recunoaște toate liniile care încep cu #, urmat de orice caractere până la \n)
- **e$ – recunoaște e doar dacă ea este la sfârșit de linie sau șir.** $ în sine nu consumă niciun caracter, ci doar ancorează ER în acea poziție (ex: ;$ recunoaște toate ; care apar la sfârșit de linie)

Modificatori pentru ER

- Dacă o ER este pusă între /.../, la sfârșit pot fi specificate unele litere cu rol de modificatori. Aceștia vor influența aplicarea ER. Modificatori pot fi:
  - **i (insensitive)** – nu se face diferența între litere mari și mici
  - **g (global)** – caută toate aparițiile ER în text, nu doar prima apariție
  - **s (single line)** – punctul recunoaște și \n
  - **m (multi line)** – textul este considerat ca fiind format din mai multe linii și se pot aplica ^...$ pentru fiecare linie. Altfel, ^...$ ancorează la începutul și sfârșitul întregului text.
  - **x (extended)** – nu se ține cont de spații în ER. Astfel ER se poate scrie mai lizibil, iar dacă este nevoie de spații, ele se vor pune în []
- ex: **/ana/ig** – recunoaște toate aparițiile cuvântului *ana* în text, indiferent de literele cu care este scris

Capturi și referințe la ele

- În ER parantezele, pe lângă rolul de a schimba ordinea operațiilor, au și rolul de captura ER din interiorul lor
- Această captură se poate folosi ulterior cu **\0..9** (*backslash urmat de o cifră*). Cifra indică la a câta paranteză deschisă se face referire, începând cu **1**. **\0** înseamnă toată ER.
- Captura este foarte utilă pentru extragerea informațiilor utile din datele de intrare sau pentru operații de genul search/replace.
- **Exemplu**: dacă avem un fișier CSV cu linii cu formatul

  nume,email,data_nastere

  putem folosi o ER de forma `^([^,]+),([^,]+),([^,]+)$` pentru a extrage numele în **\1**, emailul în **\2** și data nașterii în **\3**.

Ordinea operațiilor

- Prima oară se execută operatorii postfixați: * + ? {}
- Ulterior se execută secvențele: $\mathbf{e_1 e_2}$
- În final se execută alternativele: $\mathbf{e_1 | e_2}$
- Pentru a se modifica ordinea operațiilor, se folosesc parantezele
- Se va ține cont de faptul că dacă o ER a fost îndeplinită complet pe o anumită ramură, nu se vor mai testa și celelalte ramuri, deși s-ar putea ca și ele să fie adevărate. Din acest motiv, alternativele cele mai lungi se vor pune primele. În caz contrar, ER se va îndeplini prima oară pe o ramură mai scurtă și nu va consuma toate caracterele pe care ar trebui să le consume.
  Ex: pentru șirul de intrare "*ionescu citește*", ER ion|ionescu va recunoaște doar subșirul "*ion*", pe când ER ionescu|ion va recunoaște subșirul "*ionescu*"

Comportamentul *greedy*

- Implicit operatorii din ER încearcă să consume cât mai multe caractere (au comportament *greedy=lacom*)
- Dacă sunt mai multe caractere la fel și vrem să ne oprim la primul dintre ele, atunci va trebui să scriem ER în așa fel încât caracterul de final să nu facă parte din repetiție
- **Exemplu**: se dă propoziția

  "and", "or" și "not" sunt cuvinte în engleză

  și dorim să scriem o ER care să recunoască fiecare cuvânt dintre ghilimele
- **Greșit**: `".*"` – va recunoaște tot subșirul de la primele până la ultimele ghilimele (*"and", "or" și "not"*). Deoarece se încearcă să se consume cât de mult posibil, iar punctul se poate substitui inclusiv cu ghilimele, se vor consuma și ghilimele intermediare.
- **Corect**: `"[^"]*"` – după ghilimelele inițiale se vor consuma doar caractere care nu sunt ghilimele, până când se vor întâlni următoarele ghilimele

Exemple

- sun|mon|tue|wed|thu|fri|sat
- 0?[1-9]|[12][0-9]|3[01]
- [+-]?[0-9]+(\.[0-9]+)?
- \/\/[^\n\r\0]*
- ^[\t]*([^, \t]+)[\t]*,[\t]*([0-9]+)[\t]*$

Exemplu în C++ (1)

- Considerăm că avem un fișier CSV, numit *persons.csv*, care conține linii de forma: *nume_persoana,zi/luna/an*
- Fișierul mai poate conține și linii care sunt goale sau au conținut în alt format – aceste linii trebuie ignorate
- Numele poate conține litere mari, mici și diverse alte caractere gen cratimă sau apostrof
- La început și sfârșit de linie, precum și în jurul virgulei pot să fie spații sau taburi care trebuie ignorate
- Programul va trebui să afișeze toate numele persoanelor care au ziua de naștere în luna curentă, precum și vârsta pe care o împlinesc

```
        Ion     ,    23/10/1990

; Ana s-a nascut la Brasov
        Ana,4/3/1992

Maria  ,9/10/2001
```

```
10/2019
Ion implineste 29 ani
Maria implineste 18 ani
```

Exemplu în C++ (2)

```
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <regex>
using namespace std;
int main(){
    time_t tt=time(nullptr);
    struct tm *t=localtime(&tt);
    int crtMonth=t->tm_mon+1,crtYear=t->tm_year+1900;
    cout<<crtMonth<<'/'<<crtYear<<'\n';
    ifstream fis("persons.csv");
    string line;
    regex e("[ \t]*([^, \t]+)[ \t]*,[ \t]*[0-9]{1,2}\\/([0-9]{1,2})\\/([0-9]{1,4})[ \t]*$");
    while(getline(fis,line)){
        smatch captures;
        if(regex_match(line,captures,e)){
            int month=atoi(captures[2].str().c_str());
            int year=atoi(captures[3].str().c_str());
            if(month==crtMonth){
                cout<<captures[1]<<" implineste "<<(crtYear-year)<<" ani\n";
                }
            }
        }
    return 0;
    }
```