Beata Jankowska
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

VII. Wyrażenia regularne i klasa regex

1. Wyrażenie regularne i klasa `regex`

1.1. Wyrażenia regularne to wzorce (schematy) opisujące języki regularne. Języki te stanowią podgrupę języków formalnych i jako takie mają postać podzbiorów zbiorów wszystkich słów zbudowanych nad pewnymi alfabetami terminalnymi. Formalnie, w celu podania definicji języka regularnego, formułuje się gramatykę regularną stanowiącą generator tego języka. Dla każdego języka regularnego zadanego za pomocą gramatyki lewostronnej można też skonstruować akceptor w postaci automatu skończonego, akceptującego wszystkie i tylko słowa, które należą do danego języka.

1.2. W klasyfikacji gramatyk generacyjnych Noama Chomsky'ego, gramatyki (i języki) regularne stanowią klasę o numerze 3:

- 0 – gramatyki rekurencyjnie przeliczalne, w których wszystkie produkcje są postaci: $\alpha \rightarrow \beta$, gdzie α i β są dowolnym słowami;
- 1– gramatyki kontekstowe, w których wszystkie produkcje mają postać $\alpha\ A\ \beta \rightarrow \alpha\ \gamma\ \beta$, gdzie α i β są dowolnym słowami, A jest dowolnym symbolem nieterminalnym, a γ – dowolnym słowem niepustym;
- 2 – gramatyki bezkontekstowe, w których wszystkie produkcje są postaci $A \rightarrow \alpha$, gdzie A oznacza dowolny symbol nieterminalny, a α – dowolne słowo;
- 3 – gramatyki regularne, w których wszystkie produkcje przyjmują postać $A \rightarrow \alpha$, gdzie A oznacza dowolny symbol nieterminalny, zaś α – dowolne słowo zawierające co najwyżej jeden symbol nieterminalny znajdujący się na początku (gramatyki lewostronne liniowe) lub na końcu (gramatyki prawostronne liniowe) tego słowa.

Dla dowolnego i, $1 \leq i \leq 3$, zbiór języków generowanych przez gramatyki klasy i stanowi podzbiór właściwy zbioru języków generowanych przez gramatyki klasy (i-1).

1.3. Przykład1. Rozważmy alfabet symboli terminalnych $\Sigma = \{a, b, c\}$ oraz zbudowany nad tym alfabetem (nieskończony) język regularny J_1, w postaci:

$J_1 = \{a^k b^m c^n | \; k, m, n \in \mathrm{N}\}$.

Do języka J_1 należą, m.in., słowa: $aaabcc$, $abccccc$, $aabbbbc$.

Do zdefiniowania języka J_1 można się posłużyć:

- wyrażeniem regularnym: $a+\; b+\; c+$;
- gramatyką lewostronną regularną: $G = \langle N, \Sigma, P, C \rangle$, gdzie

  N oznacza zbiór symboli nieterminalnych, $N = \{A, B, C\}$,
  Σ oznacza zbiór symboli terminalnych $\Sigma = \{a, b, c\}$,
  P oznacza zbiór produkcji:
  $P = \{C \to Cc, C \to Bc, B \to Bb, B \to Ab, A \to Aa, A \to a\}$,
  C jest symbolem początkowym gramatyki ;
- automatem skończonym $M = \langle Q, \Sigma, \delta, s_0, F \rangle$, gdzie

  Q oznacza zbiór stanów, $Q = \{s_0, s_1, s_2, s_3\}$,
  Σ oznacza zbiór symboli terminalnych, $\Sigma = \{a, b, c\}$,
  δ oznacza funkcję przejść, $\delta: Q \times \Sigma \to Q$, zdefiniowaną jak następuje:

  $\delta(s_0, a) = s_0$,
  $\delta(s_0, a) = s_1$,
  $\delta(s_1, b) = s_1$,
  $\delta(s_1, b) = s_2$,
  $\delta(s_2, c) = s_2$,
  $\delta(s_2, c) = s_3$,

  s_0 jest stanem początkowym automatu,

  F jest zbiorem stanów końcowych automatu, $F = \{ s_3\}$.

1.4. Przykład2. Rozważmy alfabet symboli terminalnych $\Sigma = \{a, b, c\}$ oraz zbudowany nad tym alfabetem (nieskończony) język J_2, w postaci:

$J_2 = \{a^k b^m c^n | \, k, m, n \in \mathrm{N} \, ; \, k = m\}$.

Języka J_2 nie da się zdefiniować przy użyciu wyrażenia regularnego (gramatyki regularnej). Można go zdefiniować przy użyciu gramatyki klasy 2 (gramatyki bezkontekstowej).

1.5. Wyrażenia regularne są bardzo przydatne w rozwiązywaniu problemów polegających na analizie i przetwarzaniu tekstów. Z tego powodu, do wersji języka C++11 włączono bibliotekę `boost::regex`, która zawiera szereg klas ułatwiających posługiwanie się tymi wyrażeniami. W szczególności:

- `basic_regex` jest klasą szablonową dla obiektów – wyrażeń regularnych, w której parametrem jest typ znaku;
- `regex` – stanowi ukonkretnienie tej klasy dla parametru – znaku `char`,
- `wregex` – ukonkretnienie dla parametru – znaku `wchar_t`, (konstruktor klasy `regex/wregex` przyjmuje jako argument napis definiujący wyrażenie regularne),
- `basic_regex::empty()` – jest metodą boolowską, przyjmującą wartość `false` w wypadku, gdy obiekt wyrażenia regularnego został prawidłowo zainicjowany,
- `regex::regex_match` – jest metodą o:
  - dwóch argumentach obowiązkowych, badającą, czy napis reprezentowany argumentem pierwszym jest przykładem wyrażenia regularnego reprezentowanego argumentem drugim;
  - trzech argumentach obowiązkowych, badającą, czy napis reprezentowany argumentem pierwszym jest przykładem wyrażenia regularnego reprezentowanego argumentem trzecim; argument drugi ma charakter referencji do obiektu typu generycznego `match_results`, przechowującego wyniki przeprowadzonego porównania;

– `regex::regex_search` – jest metodą o dwóch argumentach, badającą, czy napis reprezentowany argumentem pierwszym mieści w sobie napis będący przykładem wyrażenia regularnego reprezentowanego argumentem drugim,
– `regex::regex_replace` – jest trójargumentową metodą dokonującą zastąpienia w napisie wejściowym reprezentowanym argumentem pierwszym „podnapisu" pasującego do wzorca reprezentowanego argumentem drugim – napisem reprezentowanym argumentem trzecim.

1.6. Oto zestaw podstawowych symboli stosowanych do konstrukcji wyrażeń regularnych w języku C++:
– `^`, `$`,
– `.`,
– `[abc]`,
– `[a-c]`,
– `[^a-c]`,
– `[[:digit:]]` lub `\d`,
– `[[:alpha:]]`,
– `[[:space:]]` lub `\s`,
– `*`, `+`, `?`,
– `{m,n}`,
– `(reg)`.

1.7. Rozważmy przykład programu przeznaczonego do zliczania we wskazanym pliku wejściowym wierszy, które mają postać zgodną z zadanym wyrażeniem regularnym:

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <regex>
#include <string>
using namespace std;
int main()
{
    int counter = 0;
    fstream file;
    file.open("Q:\\ankietaformat1.tsv", ios::in);
    string input;
    regex record("^.*22.+sporty letnie.*$");
    while (!file.eof())
    {
        std::getline(file, input);
        cout << input << endl;
        if (regex_match(input, record))
        {
            cout << "jest lancuch" << endl; counter++;
        }
        else
        {
            cout << "nie ma lancucha" << endl;
        }
    }
    cout << counter << endl << endl << endl;
}
```

1.6. Kolejny przykładowy program odpowiada na pytanie, czy zadany konkretny łańcuch wejściowy (`input`) da się dopasować do zadanego wyrażeniem regularnym wzorca (`record`). W wypadku odpowiedzi pozytywnej, na standardowe wyjście jest wyprowadzany łańcuch znaków pasujący do drugiego (`sm[2]`) wyodrębnionego (przy użyciu nawiasów `()`) członu wyrażenia regularnego.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <regex>
#include <string>
using namespace std;
int main()
{
    string input("Plec: Mezczyzna  Wiek: 22   Sporty1:
                {zimowe, letnie}x ");
    smatch sm;
    regex record("^(.*22).+Sporty1:.*(\\{.*)$");
    {
        cout << input << endl;
        if (regex_match(input, sm, record))
        {
            cout << "Jest poszukiwany lancuch" << endl;
            if (sm.ready()) cout << sm[2] << endl;
        }
        else
        {
            cout << "Nie ma poszukiwanego lancucha"
                 << endl;
        }
    }
}
```

1.7. Poniższy przykład stanowi ilustrację użycia metod `regex_search` i `regex_replace`. W wypadku, gdy zadany konkretny łańcuch wejściowy (`input`) zawiera podłańcuch pasujący do zadanego wyrażeniem regularnym wzorca (`subrecord`), to ten podłańcuch należy zastąpić innym, wskazanym za pomocą trzeciego argumentu metody `regex_replace`.

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <regex>
#include <string>
using namespace std;
int main()
{
    string input("Plec: Mezczyzna  Wiek: 22
                  Sporty1: {zimowe, letnie}x ");
    smatch sm;
    regex subrecord("22|23|24");
    cout << "Przed zmiana: "<< input << endl;
    if (regex_search(input, subrecord))
    {
        cout << "Dokonano zastapienia poszukiwanego
                 lancucha" << endl;
        cout << regex_replace(input, subrecord, "22-24")
             << endl;
    }
    else
    {
        cout << "Nie ma poszukiwanego lancucha" << endl;
    }
}
```