ONP – There and Back Again

#operator #działanie
#notacja_infiksowa
#operand
#onp #rpn
#odwrotna_notacja_polska
#continue #isalpha
#substr

Przywyczailiśmy się, że wyrażenia arytmetyczne, na przykład dodawanie, zapisujemy w poniższy sposób:

$$a + b$$

W środku mamy operator działania, w tym przypadku znak +, natomiast po obu jego stronach mamy składniki sumy. W przypadku mnożenia będą to czynniki, w przypadku dzielenia: dzielna i dzielnik, i tak dalej.* Operator zapisujemy w środku i jest to tak zwana *notacja infiksowa*, zwana też *notacją wrostkową*.† Użycie takiej notacji niejako wymusza używanie nawiasów, aby zapewnić odpowiednią kolejność obliczeń, na przykład jeśli chcemy sumę zmiennych a i b przemnożyć przez sumę zmiennych c i d, musimy zapisać to tak:

$$(a + b) \cdot (c + d)$$

Nie jest to jedyny sposób zapisu wyrażeń arytmetycznych. Prawie 100 lat temu wielki polski matematyk i filozof Jan Łukasiewicz‡ wymyślił tak zwaną Notację Polską, w której operator zapisywało się na początku działania, czyli że dodawanie powinno wyglądać tak:

$$+\, a\, b$$

To taka rewolucja? Ano tak. Bo jak wygląda w tej notacji to bardziej złożone działanie przytoczone powyżej? Otóż tak:

$$*\, +\, a\, b\, +\, c\, d$$

Brakuje czegoś, prawda? Nawiasów! W tej notacji nawiasy nie są potrzebne.

W latach 50. XX wieku okazało się, że po pewnej modyfikacji, notacja Łukasiewicza znajduje szerokie zastosowanie w technologii cyfrowej, ze względu na znaczne oszczędności w pamięci komputerów. A na czym polegała ta modyfikacja? Na tym, że operator działania zapisywało się na końcu wyrażenia, tak więc zwykłe dodawanie winno wyglądać tak:

$$a\, b\, +$$

zaś ten drugi przykład z mnożeniem dwóch sum tak powinien się prezentować:

$$a\, b\, +\, c\, d\, +\, *$$

*Ogólnie „uczestników” działania nazywa się *operandami*.
†Trudno, nie ma żadnej ładnej nazwy na ten sposób zapisu działań.
‡Nie mylić z Ignacym Łukasiewiczem, tym od lampy naftowej.

I tak narodziła się Odwrotna Notacja Polska (ang. **Reverse Polish Notation**, wymawiaj: *rewers poulisz notejszyn*, akcenty na drugim *e* w słowie **Reverse** oraz na *o* w słowie **Polish**), używana do dzisiaj.

W sposób jawny wykorzystuje się ją na przykład w języku Postscript – profesjonalnym języku opisu strony, a nawet przez sporo lat wykorzystywana była w kalkulatorach naukowych firmy Hewlett-Packard.[§]

Generalnie funkcjonowanie tej notacji jest związane z wykorzystaniem stosu, co pokażemy w dalszych częściach tego podrozdziału.

ONP – . . . and Back Again

Zaczniemy od końca, bo tak łatwiej. Napiszemy program, który tłumaczy ONP na zwykłą notację. Jak zwykle, pójdziemy na łatwiznę: założymy, że w wyrażeniu występują tylko zmienne (identyfikatory jednoliterowe, małe litery alfabetu łacińskiego), oraz że wszystkie operatory są dwuargumentowe (czyli nie ma na przykład operatora zmiany znaku). Jako przykład wybierzemy sobie wyrażenie:

```
ab+cde++*f/
```

Dla prostoty algorytmu zrezygnujemy tym razem z odstępów między identyfikatorami i operatorami.

Ciekawe, kto na tym etapie prawidłowo przewidzi postać tego wyrażenia w normalnej notacji?

Słowo się rzekło, budujemy stos, będziemy na nim odkładać kolejne elementy wyrażenia. Zasady konstrukcji są proste:

- Jeśli kolejny element wyrażenia to zmienna, wstawiamy ją na stos.
- Jeśli kolejny element wyrażenia to operator, wtedy zdejmujemy dwa wierzchnie elementy ze stosu, wstawiamy pomiędzy nie operator, otaczamy nawiasami i ponownie wstawiamy na stos.

I tak, aż do końca wyrażenia. Na koniec wypisujemy wierzchni element ze stosu – to będzie oczekiwane wyrażenie w normalnej notacji.

Allons-y! – jak mawiali starożytni Gallowie, w tym Astérix. Rozpoczynamy nasz program od deklaracji potrzebnych zmiennych i wczytania danych (czyli wyrażenia arytmetycznego zapisanego zgodnie z ONP):

```
#include <bits/stdc++.h>
using namespace std;
```

[§]Ongiś były to prawdziwe Rolls-Royce'y przenośnych urządzeń obliczeniowych – mówimy o latach 80. i 90. XX wieku. Podobno producent pozwalał zrzucić taki kalkulator z wysokości jednego metra na beton – spróbujcie zrobić to ze współczesnym smartfonem! Kocurro pamięta taką sytuację ze studiów, gdy podczas ćwiczeń z podstaw fizyki jego kolega zapomniał kalkulatora i profesor uczynnie pożyczył mu własny – właśnie HP. Niestety nie znając OPN kolega nie powalczył z takim urządzeniem, brakowało na nim przycisku `=`. . .

```
int main()
{
  string s;
  cin >> s;
  stack<string> T;
  . . .
```

Na stosie T będziemy odkładać kolejne fragmenty przetworzonego wyrażenia, zatem jego elementy powinny być typu `string`.

Teraz czeka nas spacer po kolejnych znakach ciągu s:

```
for(char c : s)
{
  . . .
```

Jeśli znak c będzie literą, wtedy oznacza zmienną (wstawiamy go na stos). Możemy to sprawdzić za pomocą bibliotecznej funkcji `isalpha()` – wszelkie inne znaki to operatory działań, które traktujemy zgodnie z tą drugą przytoczoną wyżej regułą.

Jednak jest problem z tym wstawianiem na stos: pojedynczy znak jest typu `char`, a stos musi mieć elementy typu `string`. W języku C++ te typy są traktowane jako odmienne i nie można pojedynczego znaku wepchnąć na taki stos.[¶] Prostym obejściem tego problemu jest zadeklarowanie pomocniczej zmiennej *help* typu string i nadanie jej wartości równej znakowi c:

```
if(isalpha(c))
{
  string help = c;
  T.push(help);
}
else
{
  . . .
}
```

Po instrukcji `else` musi nastąpić obsługa operatora reprezentowanego przez znak c. Zgodnie z wytycznymi, ściągamy ze stosu dwa elementy, łączymy je operatorem, otulamy nawiasami i sru na stos:

```
string x = T.top(); T.pop();
string y = T.top(); T.pop();
T.push('(' + x + c + y + ')');
```

[¶]To nie Python, tutaj ten numer nie przejdzie.

Dobrze to jest? Niestety nie. Wie ktoś, dlaczego? No to niech Czytelnik pomyśli, a tymczasem pokażemy poprawną wersję tego fragmentu kodu:

```
string x = T.top(); T.pop();
string y = T.top(); T.pop();
T.push('(' + y + c + x + ')');
```

Po zakończeniu pętli `for` na stosie pozostanie jeden element – wyrażenie w normalnej notacji. Trzeba je wypisać na ekranie, może tak:

```
s = T.top(); T.pop();
cout << s << '\n';
```

Posłużyliśmy się tutaj zmienną s, która w pierwotnej postaci nie jest nam już potrzebna.

No prawie że doskonale, ale jednak wydruk będzie miał drobną skazę: wyrażenie będą otaczać nadprogramowe nawiasy, które trzeba usunąć. Wystarczy wypisać nie cały ciąg, ale jego *podciąg* (ang. **substring**, wymawiaj: *sabstring*), pomijając początkowy i końcowy znak. Mamy do dyspozycji biblioteczną funkcję `substr()`, która oczekuje dwóch argumentów: od którego znaku zacząć (jak zawsze, numerujemy od zera) i ile znaków wziąć (w naszym przypadku długość ciągu s – tej jego nowej postaci – pomniejszoną o 2). Czyli powinno być tak:

```
cout << s.substr(1, s.size() - 2) << '\n';
```

Trochę tu się działo, więc przytoczymy cały program od początku do końca.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  string s;
  cin >> s;
  stack<string> T;
  for(char c : s)
    if(isalpha(c))
    {
      string help = c;
      T.push(help);
    }
    else
    {
      string x = T.top(); T.pop();
      string y = T.top(); T.pop();
      T.push('(' + y + c + x +')');
    }
  s = T.top(); T.pop();
  cout << s.substr(1, s.size() - 2) << '\n';
  return 0;
}
```

Po uruchomieniu programu i wklepaniu (lub wklejeniu) przykładowego wyrażenia:

```
ab+cde++*f/
```

otrzymamy wynik:

```
((a+b)*(c+(d+e)))/f
```

I tak miało być.

ONP – There...

A teraz spróbujemy w drugą stronę, troszkę trudniej będzie, ale nie panikujmy. Naszym zadaniem jest przetworzenie wyrażenia w normalnej notacji do postaci ONP. Oryginalne wyrażenie wczytamy do zmiennej globalnej s i będziemy konstruować wynik przy pomocy rekurencyjnej funkcji `parse()` (z ang. *analizować*, wymawiaj: *pars*). Kolejne instancje funkcji otrzymają jako argument przedział od-do: numer znaku, od którego trzeba zacząć (`start`) i numer znaku, przed którym trzeba skończyć (`finish`).[‖]

Poczynimy jednak spore upraszczające założenia: wszystkie działania będą odpowiednio wyposażone w nawiasy, czyli na przykład zamiast domyślnego:[**]

$$a + b + c$$

będziemy mieć:

$$(a + b) + c \qquad \text{albo} \qquad a + (b + c).$$

Ponadto nie będziemy wykorzystywać priorytetów działań, czyli zamiast:

$$a \cdot b + c$$

będziemy mieć:

$$(a \cdot b) + c$$

No to teraz powinno pójść już łatwo:

```
string s;

void parse(int start, int finish)
{
  . . .
}
```

Czy Czytelnik pamięta rozdział o rekurencji? Taką funkcję należy rozpocząć od warunku zakończenia zagnieżdżania funkcji. Zaraz sprecyzujemy, o co nam chodzi.

[‖] Jak typowe operacje w bibliotece STL.

[**] W normalnym zapisie opuszczenie nawiasów jest tutaj jak najbardziej dozwolone, bo dodawanie jest działaniem *łącznym*. Pamiętacie takie hasło ze szkoły powszechnej?

Typowe wyrażenie arytmetyczne wygląda tak:

coś operator coś_innego

Ale to wcale nie jest najprostsze wyrażenie arytmetyczne: bo takie minimalistyczne, to po prostu:

coś

a to coś to w wersji minimum jednoliterowy identyfikator, po prostu jeden znak, tak jest w naszej bajce. (Kocurro lubi tę bajkę.) I to jest nasz warunek zakończenia rekurencji: jeśli funkcja `parse()` dostanie do przetworzenia jednoznakowy wycinek ciągu s, to nie ma dyskutować, tylko wiosłować, *pardon*, wypisać go na ekranie i zakończyć działanie (mówimy o konkretnej instancji tej funkcji). Możemy zapisać to tak:

```
if(start == finish - 1)
{
  cout << s[start];
  return;
}
```

Instrukcja `return` gwarantuje wyjście z funkcji i nieprzetwarzanie reszty jej kodu.

Co dalej? No, teraz mamy zasadniczo dwie możliwości. Albo wyrażenie postaci takiej:

id operator coś

gdzie *id* oznacza jednoliterowy identyfikator, albo wyrażenie zaczynające się od nawiasu otwierającego i zawierające *bógwico*, a potem może ewentualnie nastąpić operator i znów *bógwico*, tylko inne, czyli tak:

(*bógwico*)

względnie tak:

(*bógwico*) *operator bógwico_inne*

Musimy być przygotowani na każdą możliwość. Zacznijmy od sytuacji, gdy mamy identyfikator na początku: wariant *id*/*operator*/*coś*. Obczaimy to łatwo: jeśli na pozycji *start* stoi litera, to jest właśnie ten wariant – sprawdzimy to funkcją `isalpha()`. Zaraz po tym identyfikatorze musi być operator, a potem dalsza część wyrażenia, zaczynająca się od pozycji $start + 2$. Jeśli mamy to przepisać zgodnie z ONP, to najpierw wypiszemy wspomniany identyfikator, potem tę dalszą część wyrażenia, a na końcu operator:

```
if(isalpha(s[start]))
{
  cout << s[start];
  parse(start + 2, finish);
  cout << s[start + 1];
  return;
}
```

Konsekwentnie używamy instrukcji `return`, aby „odcinać” kolejne warianty, a nie zagnieżdżać kolejne poziomy instrukcji warunkowej.

Teraz musimy uporać się z wariantem wyrażenia, które zaczyna się od nawiasu otwierającego. Musimy znaleźć położenie pasującego do niego nawiasu zamykającego i powinniśmy to zrobić szybko i sprawnie. Nie ma sensu teraz przeglądać ciągu znaków s, bo to oznaczałoby wielokrotne spacerowanie po tablicy znaków (w przypadku zagnieżdżonych nawiasów). Najlepiej będzie już na etapie wczytywania danych oznaczyć sobie, gdzie występują pasujące do siebie nawiasy. Wystarczy do tego zwykły wektor liczb całkowitych (P), zadeklarowany jako zmienna globalna przed definicją funkcji `parse()`. Konieczne będzie podanie jego rozmiaru, powiedzmy 1000 elementów, to znaczy, że taka będzie maksymalna długość wczytywanego wyrażenia. W wektorze będą wykorzystane tylko te elementy, których numery odpowiadają pozycjom nawiasów otwierających.

Przyda się również stos, o czym chyba pamiętamy z podrozdziału *Nawiasy*. Wczytany ciąg s przeglądamy znak po znaku: jeśli trafimy na nawias otwierajacy, odkładamy jego pozycję na stos, a jeśli trafimy na nawias zamykający, zapisujemy jego pozycję w wektorze. Wyszukiwanie nawiasów powinniśmy zrealizować przed pierwszym wywołaniem funkcji `parse()`. Funkcja `main()` mogłaby zatem wyglądać tak:

```
int main()
{
  cin >> s;
  stack<int> Q;
  for(int i = 0; i < s.size(); i++)
    if(s[i] == '(')
      Q.push(i);
    else if(s[i] == ')')
    {
      P[Q.top()] = i;
      Q.pop();
    }
  parse(0, s.size());
  cout << '\n';
  return 0;
}
```

Teraz możemy już uzupełnić końcówkę funkcji `parse()`:

```
  parse(start + 1, P[start]);
  if(P[start] < finish - 1)
  {
    parse(P[start] + 2, finish);
    cout << s[P[start] + 1];
  }
}
```

Pierwsze wywołanie potomka funkcji `parse()` dotyczy fragmentu wyrażenia wewnątrz nawiasu. Nawias zamykający znajduje się na pozycji $P[start]$.[††]

[††]Pamiętamy, że funkcja `parse(n, m)` analizuje znaki wyrażenia o numerach od n do $m-1$.

Powyższa instrukcja warunkowa sprawdza, czy po tym nawiasie zamykającym jeszcze coś jest i w razie potrzeby wywołuje potomka funkcji dla tej drugiej części wyrażenia, zaś na samym końcu wypisuje operator, który znajduje się na pozycji $P[start] + 1$.

Ponieważ pisaliśmy funkcję `parse()` w odcinkach, przytoczymy teraz całą jej treść:

```
string s;
vector<int> P(1000);

void parse(int start, int finish)
{
  if(start == finish - 1)
  {
    cout << s[start];
    return;
  }
  if(isalpha(s[start]))
  {
    cout << s[start];
    parse(start + 2, finish);
    cout << s[start + 1];
    return;
  }
  parse(start + 1, P[start]);
  if(P[start] < finish - 1)
  {
    parse(P[start] + 2, finish);
    cout << s[P[start] + 1];
  }
}
```

Jeśli uruchomimy nasz program i wpiszemy wyrażenie:

```
((a+b)*(c+(d+e)))/f
```

otrzymamy jego wersję w ONP:

```
ab+cde++*f/
```

I to byłoby na tyle.