Programowanie, część III

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

29 marca 2011

Problem z układem współrzędnych

Rysując wykres funkcji zwykle jesteśmy zainteresowani określonym zakresem wartości dla zmiennej niezależnej (od $x0$ do $x1$) oraz pewnym zakresem wartości funkcji (od $y0$ do $y1$).

Problem z układem współrzędnych

Rysując wykres funkcji zwykle jesteśmy zainteresowani określonym zakresem wartości dla zmiennej niezależnej (od $x0$ do $x1$) oraz pewnym zakresem wartości funkcji (od $y0$ do $y1$).

Odpowiada to prostokątnemu podzbiorowi płaszczyzny z kartezjańskim układem współrzędnych. Tymczasem do dyspozycji mamy prostokątny obszar ekranu z innym układem współrzędnych.

Problem z układem współrzędnych

Rysując wykres funkcji zwykle jesteśmy zainteresowani określonym zakresem wartości dla zmiennej niezależnej (od $x0$ do $x1$) oraz pewnym zakresem wartości funkcji (od $y0$ do $y1$).

Odpowiada to prostokątnemu podzbiorowi płaszczyzny z kartezjańskim układem współrzędnych. Tymczasem do dyspozycji mamy prostokątny obszar ekranu z innym układem współrzędnych.

Przed narysowaniem wykresu najwygodniej jest przetransformować „ekranowy" układ współrzędnych tak, aby współrzędne w obszarze, który mamy do dyspozycji, odpowiadały zakresom wartości dla zmiennej niezależnej (od $x0$ do $x1$) i dla funkcji (od $y0$ do $y1$).

Transformacja układu współrzędnych

Skalowanie

Układ współrzędnych związany z obszarem rysunku skalujemy tak, aby zakres wartości dla współrzędnej x był odcinkiem o długości $x1 - x0$, a zakres wartości dla współrzędnej y był odcinkiem o długości $y1 - y0$. Ponadto zmieniamy znak y tak, aby oś była skierowana w górę:

```
QPainter painter(this);

painter.scale(width() / (x1 - x0), -height() / (y1 - y0));
```

Transformacja układu współrzędnych

Skalowanie

Układ współrzędnych związany z obszarem rysunku skalujemy tak, aby zakres wartości dla współrzędnej x był odcinkiem o długości $x1 - x0$, a zakres wartości dla współrzędnej y był odcinkiem o długości $y1 - y0$. Ponadto zmieniamy znak y tak, aby oś była skierowana w górę:

```
QPainter painter(this);

painter.scale(width() / (x1 - x0), -height() / (y1 - y0));
```

Translacja

Początek układu współrzędnych przesuwamy do punktu $(x0, y0)$, biorąc pod uwagę to, że przed translacją lewy górny róg okna ma współrzędne $(0, 0)$, a chcemy, żeby miał współrzędne $(x0, y1)$:

```
painter.translate(-x0, -y1);
```

Rysowanie wykresu

Przed narysowaniem wykresu przeprowadzamy transformację układu współrzędnych omówioną wcześniej.

Rysowanie wykresu

Przed narysowaniem wykresu przeprowadzamy transformację układu współrzędnych omówioną wcześniej.

Wykres rysujemy jako linię łamaną, obliczając wartości funkcji dla wybranego zbioru wartości zmiennej niezależnej $x_j \in [x0, x1]$ (np. w n symetrycznie rozłożonych punktach) i łącząc odcinkami linii prostej punkty w wpółrzędnych $(x_j, f(x_j))$, np.:

```
double r = -M_PI;
double krok = M_PI / 100;  // Funkcję obliczamy 101 razy.
QPointF p1(r, sin(r));
QPointF p2;
for (r += krok; r <= M_PI; r += krok) {
    p2.setX(r);
    p2.setY(sin(r));
    painter.drawLine(p1, p2);
    p1 = p2;
}
```

Rysowanie podziałki i etykiet liczbowych

Do wykresu należy dołączyć podziałkę z informacją dla jakich wartości x i y narysowana jest funkcja.

Rysowanie podziałki i etykiet liczbowych

Do wykresu należy dołączyć podziałkę z informacją dla jakich wartości x i y narysowana jest funkcja.

W tym celu w odpowiednich miejscach rysunku trzeba umieścić odcinki linii prostej reprezentujące podziałkę i napisy reprezentujące odpowiednie wartości liczbowe.

Rysowanie podziałki i etykiet liczbowych

Do wykresu należy dołączyć podziałkę z informacją dla jakich wartości x i y narysowana jest funkcja.

W tym celu w odpowiednich miejscach rysunku trzeba umieścić odcinki linii prostej reprezentujące podziałkę i napisy reprezentujące odpowiednie wartości liczbowe.

Ponieważ napisy te znajdują się zwykle na zewnątrz obszaru zawierającego wykres funkcji, więc najwygodniej jest narysować je przed przeskalowaniem układu współrzędnych.

Rysowanie podziałki i etykiet liczbowych

Do wykresu należy dołączyć podziałkę z informacją dla jakich wartości x i y narysowana jest funkcja.

W tym celu w odpowiednich miejscach rysunku trzeba umieścić odcinki linii prostej reprezentujące podziałkę i napisy reprezentujące odpowiednie wartości liczbowe.

Ponieważ napisy te znajdują się zwykle na zewnątrz obszaru zawierającego wykres funkcji, więc najwygodniej jest narysować je przed przeskalowaniem układu współrzędnych.

Ponadto trzeba uwzględnić ich rozmiary tak, aby obszar zawierający wykres funkcji był odpowiednio mniejszy od dostępnego okna (tzn. część dostępnego okna przeznaczamy na narysowanie etykiet liczbowych).

Znakowe reprezentacje liczb w *Qt*

`QString::number()`

Statyczna metoda, którą można wykorzystać do otrzymania napisu odpowiadającego określonej wartości liczbowej.

Znakowe reprezentacje liczb w *Qt*

`QString::number()`

Statyczna metoda, którą można wykorzystać do otrzymania napisu odpowiadającego określonej wartości liczbowej.

W ogólności ma ona trzy argumenty:

1. Liczbę, której reprezentację znakową chcemy otrzymać.
2. Znak określający format (`'f'` oznacza ułamek dziesiętny).
3. Liczbę miejsc po przecinku (kropce).

Np. reprezentację wartości zmiennej `r` w postaci z częścią ułamkową o dwóch miejscach po przecinku (kropce) można otrzymać w następujący sposób:

```
QString::number(r, 'f', 2)
```

Drukowanie napisów

Mając napis reprezentujący liczbę można go wydrukować używając (jednego z wariantów) metody `drawText()`, np.:

```
painter.drawText(x, y, szer, wys,
                 Qt::AlignHCenter | Qt::AlignVCenter,
                 QString::number(r, 'f', 2));
```

Drukowanie napisów

Mając napis reprezentujący liczbę można go wydrukować używając (jednego z wariantów) metody `drawText()`, np.:

```
painter.drawText(x, y, szer, wys,
                 Qt::AlignHCenter | Qt::AlignVCenter,
                 QString::number(r, 'f', 2));
```

Argumenty `drawText()` oznaczają:

1. Współrzędne lewego górnego rogu prostokąta zawierającego napis.
2. Szerokość i wysokość prostokąta zawierającego napis.
3. Sposób wyrównania.
4. Napis do wydrukowania.

Obliczanie rozmiarów napisów

Mając napis reprezentujący liczbę można obliczyć współrzędne i rozmiary prostokąta, który on zajmie po wydrukowaniu, używając (jednego z wariantów) metody `boundingRect()`, np.:

```
painter.boundingRect(0, 0, width(), height(),
                     Qt::AlignHCenter | Qt::AlignVCenter,
                     QString::number(r, 'f', 2));
```

Obliczanie rozmiarów napisów

Mając napis reprezentujący liczbę można obliczyć współrzędne i rozmiary prostokąta, który on zajmie po wydrukowaniu, używając (jednego z wariantów) metody `boundingRect()`, np.:

```
painter.boundingRect(0, 0, width(), height(),
                     Qt::AlignHCenter | Qt::AlignVCenter,
                     QString::number(r, 'f', 2));
```

Argumenty `boundingRect()` oznaczają:

1. Współrzędne lewego górnego rogu obszaru zawierającego napis.
2. Szerokość i wysokość obszaru zawierającego napis.
3. Sposób wyrównania.
4. Napis do wydrukowania.

Co to jest szablon

Szablon (*ang. template*)

Konstrukcja pozwalająca uniknąć duplikowania kodu źródłowego poprzez zapisanie go w sposób umożliwiający wielokrotną kompilację dla różnych „podstawionych” klas.

Co to jest szablon

Szablon (*ang. template*)

Konstrukcja pozwalająca uniknąć duplikowania kodu źródłowego poprzez zapisanie go w sposób umożliwiający wielokrotną kompilację dla różnych „podstawionych" klas.

Następujący zapis oznacza, że definicja klasy `ElementListy` jest szablonem i należy ją dopełnić (w dalszej części programu) określając klasę `T`, dla której ma ona być skompilowana (może ona być kompilowana dla różnych klas `T`):

```
template<class T> class ElementListy {
 public:
    T dane;
    ElementListy *nast;
};
```

Przykład – klasa Stos zdefiniowana jako szablon

Na tym etapie klasa T, od której zależy definicja klasy Stos, nie została jeszcze określona.

```
template<class T> class Stos {
    ElementListy<T> *pierwszy;
 public:
    Stos(void): pierwszy(NULL) {}
    void wstaw(T s);
    T zdejmij(void);
    bool pusty(void);
};

template<class T> void Stos<T>::wstaw(T obj)
{
    ElementListy<T> *el;

    el = new ElementListy<T>;
    if (!el)
        return;
    el->dane = obj;
    el->nast = pierwszy;
    pierwszy = el;
}
```

```
template<class T> T Stos<T>::zdejmij(void)
{
    ElementListy<T> *el;
    T obj;

    // Zakładamy, że stos nie jest pusty.
    el = pierwszy;
    obj = el->dane;
    pierwszy = el->nast;
    delete el;
    return obj;
}

template<class T> bool Stos<T>::pusty(void)
{
    return pierwszy == NULL;
}
```

Przykład – program wykorzystujący klasę Stos

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cerr << "Wymagany 1 argument" << endl;
        return EXIT_FAILURE;
    }

    ifstream input(argv[1]);

    if (!input) {
        cerr << "Problem z otwieraniem pliku " << argv[1] << endl;
        return EXIT_FAILURE;
    }

    // Informujemy kompilator, że klasą T, od której zależy definicja klasy Stos, jest string.
    Stos<string> st;
    string wiersz;

    // Odczytujemy wiersze z pliku i wkładamy na stos.
    while (getline(input, wiersz))
        st.wstaw(wiersz);

    // Drukujemy zawartość stosu.
    while (!st.pusty())
        cout << st.zdejmij() << endl;

    return EXIT_SUCCESS;
}
```

STL (*ang. Standard Template Library*)

STL jest częścią biblioteki standardowej C++ zawierającą definicje klas i funkcji pozwalających na posługiwanie się typowymi strukturami danych (np. wektorami, kolejkami, stosami, drzewami), zdefiniowanymi w oparciu o szablony.

STL (*ang. Standard Template Library*)

STL jest częścią biblioteki standardowej C++ zawierającą definicje klas i funkcji pozwalających na posługiwanie się typowymi strukturami danych (np. wektorami, kolejkami, stosami, drzewami), zdefiniowanymi w oparciu o szablony.

Klasy z STL, zwane pojemnikami (*ang. container*), reprezentują najczęściej używane struktury danych i udostępniają metody ułatwiające posługiwanie się nimi (np. wstawianie i usuwanie elementów).

STL (*ang. Standard Template Library*)

STL jest częścią biblioteki standardowej C++ zawierającą definicje klas i funkcji pozwalających na posługiwanie się typowymi strukturami danych (np. wektorami, kolejkami, stosami, drzewami), zdefiniowanymi w oparciu o szablony.

Klasy z STL, zwane pojemnikami (*ang. container*), reprezentują najczęściej używane struktury danych i udostępniają metody ułatwiające posługiwanie się nimi (np. wstawianie i usuwanie elementów).

Funkcje z STL, nazywane także algorytmami (*ang. algorithm*), służą do przeprowadzania typowych operacji na strukturach danych reprezentowanych przez klasy z STL (np. sortowanie, przeszukiwanie).

Iteratory

Iterator

Obiekt reprezentujący uogólniony wskaźnik, pozwalający na określenie pozycji elementu w pewnej strukturze danych (np. na liście lub w drzewie).

Iteratory

Iterator

Obiekt reprezentujący uogólniony wskaźnik, pozwalający na określenie pozycji elementu w pewnej strukturze danych (np. na liście lub w drzewie).

Iteratory pozwalają na rozłączenie algorytmów i pojemników w taki sposób, że algorytmy są szablonami sparametryzowanymi typem iteratora (tzn. działają w sposób zależny od tego, jaki typ iteratora jest związany z pojemnikiem, na którym przeprowadzane są operacje) i nie są ograniczone do ustalonego typu pojemnika.

Iteratory

Iterator

Obiekt reprezentujący uogólniony wskaźnik, pozwalający na określenie pozycji elementu w pewnej strukturze danych (np. na liście lub w drzewie).

Iteratory pozwalają na rozłączenie algorytmów i pojemników w taki sposób, że algorytmy są szablonami sparametryzowanymi typem iteratora (tzn. działają w sposób zależny od tego, jaki typ iteratora jest związany z pojemnikiem, na którym przeprowadzane są operacje) i nie są ograniczone do ustalonego typu pojemnika.

Definicje klas dla iteratorów muszą, m. in., określać działanie operatorów pre- i postinkrementacji ++ oraz operatora * odpowiadającego wskaźnikowemu operatorowi zwracającemu referencję do wskazywanej zmiennej.

Zapis algorytmu z użyciem iteratorów – przykład

Przeszukiwanie tablicy o elemetach typu `int` z użyciem wskaźników:

```
int* find(int *start, int *koniec, const int& wart) {
    while (start != koniec && *start != wart)
        ++start;
    return start;
}
```

Zapis algorytmu z użyciem iteratorów – przykład

Przeszukiwanie tablicy o elemetach typu `int` z użyciem wskaźników:

```
int* find(int *start, int *koniec, const int& wart) {
    while (start != koniec && *start != wart)
        ++start;
    return start;
}
```

Przeszukiwanie dowolnej struktury danych o elementach dowolnego typu z uzyciem iteratorów:

```
template <class InputIterator, class T>
InputIterator find(InputIterator start, InputIterator koniec, const T& wart) {
    while (start != koniec && *start != wart)
        ++start;
    return start;
}
```

Zapis algorytmu z użyciem iteratorów – przykład

Przeszukiwanie tablicy o elemetach typu `int` z użyciem wskaźników:

```
int* find(int *start, int *koniec, const int& wart) {
    while (start != koniec && *start != wart)
        ++start;
    return start;
}
```

Przeszukiwanie dowolnej struktury danych o elementach dowolnego typu z uzyciem iteratorów:

```
template <class InputIterator, class T>
InputIterator find(InputIterator start, InputIterator koniec, const T& wart) {
    while (start != koniec && *start != wart)
        ++start;
    return start;
}
```

STL zawiera definicje klas dla iteratorów odpowiadających klasom reprezentującym różne pojemniki (np. wektor, stos, kolejkę).

Przykład – drukowanie wierszy w odwrotnej kolejności

Program drukuje w odwrotnej kolejności wiersze odczytane z pliku.

```
#include <iostream>
#include <fstream>
#include <stack>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    ifstream input(argv[1]);
    stack<string> st;
    string wiersz;

    // Odczytujemy wiersze z pliku i wkładamy na stos.
    while (getline(input, wiersz))
        st.push(wiersz);

    // Drukujemy zawartość stosu.
    while (!st.empty()) {
        cout << st.top() << endl;
        st.pop();
    }

    return EXIT_SUCCESS;
}
```

Przykład – sortowanie wierszy odczytanych z pliku

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    ifstream plik(argv[1]);
    vector<string> wiersze;
    string w;

    while (getline(plik, w))
        wiersze.push_back(w);

    // Iteratory wiersze.begin() i wiersze.end() określają pozycje pierwszego
    // i ostatniego elementu wektora (i zarazem zakres elementów do posortowania).
    sort(wiersze.begin(), wiersze.end());

    for (int i = 0; i < wiersze.size(); i++)
        cout << wiersze[i] << endl;

    return EXIT_SUCCESS;
}
```

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

ONP została zaproponowana w połowie lat 1950 przez Charlesa Hamblina, jako „odwrócenie" beznawiasowej notacji polskiej Jana Łukasiewicza.

Odwrotna notacja polska (ONP)

RPN (*ang. Reverse Polish Notation*)

ONP, zwana także notacją postfiksową (*ang. postfix*), jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach, np. $2\ \ 4\ +$ zamiast $2 + 4$.

ONP pozwala na całkowitą rezygnację z użycia nawiasów w wyrażeniach, ponieważ jednoznacznie określa kolejność wykonywanych działań.

ONP została zaproponowana w połowie lat 1950 przez Charlesa Hamblina, jako „odwrócenie" beznawiasowej notacji polskiej Jana Łukasiewicza.

Okazuje się, że algorytm obliczania wyrażeń w ONP jest stosunkowo prosty. Podobnie prosty jest algorytm przekształcania wyrażeń w notacji „tradycyjnej" (infiksowej) na ONP.

Algorytm obliczania wyrażeń w ONP

Potrzebny jest stos do przechowywania liczb.

1. Dla każdego elementu wyrażenia w ONP od lewej:
   1. Jeżeli bieżący element jest liczbą, umieść ją na stosie.
   2. Jeżeli bieżący element jest operatorem $(+, -, *, /)$, to:
      - Zdejmij dwie liczby ze stosu.
      - Przeprowadź obliczenia: druga liczba zdjęta ze stosu-operator-pierwsza liczba zdjęta ze stosu.
      - Umieść wynik na stosie.
2. Po zakończeniu pętli jedyna znajdująca się na stosie liczba jest wynikiem wyrażenia (o ile wyrażenie jest poprawne).

Obliczanie wyrażeń w ONP – przykład

Wyrażenie

$2\ 3\ +\ 4\ *\ 10\ -$

Przebieg obliczeń

Krok	**Na stosie**
1	2
2	2 3
3	5
4	5 4
5	20
6	20 10
7	10

Funkcja obliczająca wartość wyrażenia w ONP

```
double oblicz(string& wyr)
{
    stack<double> stos;
    istringstream input(wyr);
    string s;
    double r;

    while (input >> s) {
        istringstream is(s);

        if(is >> r) {
            stos.push(r);
            continue;
        }

        if (jest_operatorem(s[0])) {
            r = stos.top();
            stos.pop();

            r = wykonaj_obliczenia(r, stos.top(), s[0]);
            stos.pop();

            stos.push(r);
        }
    }
    r = stos.top();
    stos.pop();

    return r;
}


bool jest_operatorem(char c)
{
    return c == '+' || c == '-'
        || c == '*' || c == '/';
}
```

Funkcja obliczająca wartość wyrażenia w ONP (c. d.)

```
double wykonaj_obliczenia(double r, double d, char c)
{
    switch (c) {
    case '+':
        d += r;
        break;
    case '-':
        d -= r;
        break;
    case '*':
        d *= r;
        break;
    case '/':
        d /= r;
        break;
    }

    return d;
}
```

Algorytm zamiany zapisu wyrażenia na ONP (1)

Potrzebny jest stos do przechowywania symboli oraz łańcuch, w którym będzie zapisany wynik.

Algorytm zamiany zapisu wyrażenia na ONP (1)

Potrzebny jest stos do przechowywania symboli oraz łańcuch, w którym będzie zapisany wynik.

Dla każdego elementu wyrażenia w notacji infiksowej (z nawiasami) (1)

- Jeśli bieżący element jest liczbą, wstaw go oraz znak przerwy na koniec wynikowego łańcucha.
- Jeśli bieżący element jest operatorem, to:
  1. Powtarzaj, aż na wierzchołku stosu nie będzie operatora:
     - Zdejmij operator ze stosu.
     - Wstaw zdjęty ze stosu operator oraz znak przerwy na koniec wynikowego łańcucha.
  2. Wstaw bieżący element na stos.
- Jeśli bieżący element jest lewym nawiasem, wstaw go na stos.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Powtarzaj, aż stos będzie pusty

- Zdejmij symbol ze stosu.
- Wstaw symbol zdjęty ze stosu oraz znak przerwy na koniec wynikowego łańcucha.

Algorytm zamiany zapisu wyrażenia na ONP (2)

Dla każdego elementu wyrażenia w notacji infiksowej (2)

- Jeśli bieżący element jest prawym nawiasem, to:
  1. Zdejmij symbol ze stosu.
  2. Powtarzaj, dopóki zdjęty ze stosu symbol nie jest lewym nawiasem:
     - Wstaw zdjęty ze stosu symbol i znak przerwy na koniec wynikowego łańcucha.
     - Zdejmij symbol ze stosu.

Powtarzaj, aż stos będzie pusty

- Zdejmij symbol ze stosu.
- Wstaw symbol zdjęty ze stosu oraz znak przerwy na koniec wynikowego łańcucha.

Usuń ostatni znak (znak przerwy) z wynikowego łańcucha

Zamiana zapisu wyrażenia na ONP – przykład I

Przebieg obliczeń dla wyrażenia $((2+3)*5)-1$

Krok	Na stosie	Wynik
1	(	
2	((	
3	((	2
4	((+	2
5	((+	2 3
6	(	2 3 +
7	(*	2 3 +
8	(*	2 3 + 5
9		2 3 + 5 *
10	−	2 3 + 5 *
11	−	2 3 + 5 * 1
12		2 3 + 5 * 1 −

Modyfikacja związana z priorytetami operatorów

Jeżeli operatory mają różne priorytety, powyższy algorytm może nie dać prawidłowego wyniku dla wyrażenia bez nawiasów, więc trzeba go trochę zmienić.

Modyfikacja związana z priorytetami operatorów

Jeżeli operatory mają różne priorytety, powyższy algorytm może nie dać prawidłowego wyniku dla wyrażenia bez nawiasów, więc trzeba go trochę zmienić.

Dla każdego elementu wyrażenia w notacji infiksowej (z nawiasami)

- Jeśli bieżący element jest liczbą, wstaw go oraz znak przerwy na koniec wynikowego łańcucha.
- Jeśli bieżący element jest operatorem, to:
  1. Powtarzaj, aż na wierzchołku stosu nie będzie operatora o niższym lub równym priorytecie:
     - Zdejmij operator ze stosu.
     - Wstaw zdjęty ze stosu operator oraz znak przerwy na koniec wynikowego łańcucha.
  2. Wstaw bieżący element na stos.

Zamiana zapisu wyrażenia na ONP – przykład II

Przebieg obliczeń dla wyrażenia $(2 * 3 + 5)/2$

Krok	Na stosie	Wynik
1	(	
3	(	2
4	(*	2
5	(*	2 3
6	(+	2 3 *
7	(+	2 3 * 5
8		2 3 * 5 +
9	/	2 3 * 5 +
10	/	2 3 * 5 + 2
11		2 3 * 5 + 2 /

Funkcja zamieniająca zapis wyrażenia na ONP

```
string infix_postfix(string& wyr)
{
    stack<char> stos;
    istringstream input(wyr);
    string s, postfix("");

    while (input >> s) {
        istringstream is(s);
        double r;

        if(is >> r) {
            postfix += s + ' ';
            continue;
        }
        char c = s[0];
        if (jest_operatorem(c)) {
            while (!stos.empty()) {
                char znak = stos.top();
                if (!jest_operatorem(znak))
                    break;
                if (!jest_silniejszy(znak, c))
                    break;
                postfix += znak;
                postfix += ' ';
                stos.pop();
            }
            stos.push(c);
        } else if (c == '(') {
            stos.push(c);
        } else if (c == ')') {
            char znak = stos.top();
            stos.pop();
            while (znak != '(') {
                postfix += znak;
                postfix += ' ';
                znak = stos.top();
                stos.pop();
            }
        }
    }

    while (!stos.empty()) {
        postfix += stos.top();
        stos.pop();
        postfix += ' ';
    }

    if (postfix.length() > 0)
        postfix.erase(postfix.length() - 1);

    return postfix;
}
```

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

W celu uniknięcia tego założenia „produkcyjna" wersja funkcji `infix_postfix()` musi odczytywać wejściowe wyrażenie znak po znaku.

Odczytywanie wyrażenia znak po znaku

Funkcja `infix_postfix()` przedstawiona wcześniej jest zaprojektowana z założeniem, że poszczególne elementy wyrażenia (tzn. liczby, operatory, nawiasy) będą rozdzielone znakami przerwy.

W celu uniknięcia tego założenia „produkcyjna" wersja funkcji `infix_postfix()` musi odczytywać wejściowe wyrażenie znak po znaku.

Niestety powoduje to komplikację polegająca na tym, ze znak `'-'` może być interpretowany jako:

- operator dwuargumentowy,
- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie,

którą trzeba uwzględnić.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak '-' oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak `'-'` oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Wystarczy zatem wprowadzić zmienną, która będzie otrzymywać wartość `false` po odczytaniu z wyrażenia wejściowego liczby lub prawego nawiasu oraz wartość `true` po odczytaniu jakiegokolwiek innego symbolu.

Odczytywanie wyrażenia znak po znaku – odejmowanie

W tym celu można wykorzystać obserwację, że znak `'-'` oznacza operator dwuargumentowy w dwóch przypadkach:

1. Jeżeli znajduje się w wyrażeniu bezpośrednio za nawiasem zamykającym (prawym).
2. Jeżeli znajduje się w wyrażeniu bezpośrednio za liczbą.

Wystarczy zatem wprowadzić zmienną, która będzie otrzymywać wartość `false` po odczytaniu z wyrażenia wejściowego liczby lub prawego nawiasu oraz wartość `true` po odczytaniu jakiegokolwiek innego symbolu.

Wtedy, jeżeli ta zmienna ma wartość `false` i z wyrażenia zostanie odczytany znak `'-'`, to ten znak należy traktować jako operator dwuargumentowy (tzn. odejmowanie).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

W takim przypadku w wyrażeniu wynikowym (w ONP) powinien być zapisany symbol reprezentujący zmianę znaku ostatnio obliczonego wyrażenia (np. $\sim$).

Odczytywanie wyrażenia znak po znaku – zmiana znaku

W przypadkach, w których znak '-' nie reprezentuje operatora argumentowego, powinien on być interpretowany jako:

- część zapisu liczby ujemnej,
- zmiana znaku podwyrażenia w nawiasie.

Drugi przypadek ma miejsce wtedy, gdy pierwszy (nie będący przerwą) znak za znakiem '-' jest nawiasem otwierającym (lewym).

W takim przypadku w wyrażeniu wynikowym (w ONP) powinien być zapisany symbol reprezentujący zmianę znaku ostatnio obliczonego wyrażenia (np. $\sim$).

Np. dla wyrażenia $-(2+3)$ odpowiednikiem w ONP może być $2\ \ 3\ +\ \sim$

Przetwarzanie wyrażenia znak po znaku – liczby

Odczytując wejściowe wyrażenie znak po znaku należy odpowiednio uwzględniać liczby.

Przetwarzanie wyrażenia znak po znaku – liczby

Odczytując wejściowe wyrażenie znak po znaku należy odpowiednio uwzględniać liczby.

W tym celu można zastosować następujący algorytm:

1. Przypuśćmy, że znaleźliśmy znak (być może) będący początkiem liczby.
2. Znajdujemy najbliższy znak przerwy lub nawias zamykający za nim.
3. Próbujemy odczytać liczbę z wycinka łańcucha wejściowego między tymi dwoma znakami.
4. Jeżeli operacja zakończy się powodzeniem, kopiujemy ten wycinek w całości (z „doklejonym” znakiem przerwy) do wynikowego łańcucha.
5. Przechodzimy do pierwszego znaku położonego za tym wycinkiem.

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

W tym celu można wykorzystać obserwację, że w każdym przypadku zapis funkcji standardowej składa się z nazwy (1 lub więcej znaków alfanumerycznych), bezpośrednio po której znajduje się nawias otwierający (lewy).

Zamiana zapisu wyrażeń na ONP i standardowe funkcje

Wygodnie jest uwzględniać standardowe funkcje (np. $\sin(x)$, $\cos(x)$) występujące w wyrażeniach przy zamianie zapisu tych wyrażeń na ONP.

W tym celu można wykorzystać obserwację, że w każdym przypadku zapis funkcji standardowej składa się z nazwy (1 lub więcej znaków alfanumerycznych), bezpośrednio po której znajduje się nawias otwierający (lewy).

W związku z tym standardową funkcję można traktować podobnie, jak nawias otwierający poprzedzony znakiem '-', oznaczającym zmianę znaku wyrażenia, lecz zamiast symbolu reprezentującego zmianę znaku w wynikowym łańcuchu powinien być zapisywany symbol reprezentujący daną funkcje (np. s dla $\sin(x)$ lub c dla $\cos(x)$).

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Oczywiście jeśli pierwsza procedura uwzględnia zamianę znaku wyrażeń oraz funkcje standardowe, to druga procedura musi rozpoznawać oznaczające je symbole „wyprodukowane” przez pierwszą procedurę.

Łączenie zamiany zapisu wyrażeń na ONP i obliczania ich

Procedurę zamiany wyrażenia w zapisie infiksowym na ONP (zapis postfiksowy) można połączyć z procedurą obliczania wyrażeń w zapisie postfiksowym.

Wtedy łańcuch wynikowy pochodzący z pierwszej procedury staje się łańcuchem wejściowym dla drugiej procedury.

Oczywiście jeśli pierwsza procedura uwzględnia zamianę znaku wyrażeń oraz funkcje standardowe, to druga procedura musi rozpoznawać oznaczające je symbole „wyprodukowane” przez pierwszą procedurę.

Dodatkowo, na potrzeby obliczania wartości funkcji dla różnych wartości zmiennej niezależnej, należy uwzględnić zapis zmiennej niezależnej w wyrażeniu wejściowym i w ONP.

Literatura


J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4 (2nd Edition)* (Prentice Hall, Westford, 2008).


Qt 4.5 Reference Documentation, `http://doc.trolltech.com/4.5/index.html`


B. Stroustrup, *Język C++* (Wydawnictwo Naukowo-Techniczne, Warszawa 2002).


T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów* (Wydawnictwa Naukowo-Techniczne, Warszawa 2005).