1. Odwrotna notacja polska (ONP)

W znanym ci sposobie zapisywania wyrażeń algebraicznych znaki działań umieszcza się pomiędzy argumentami, a o kolejności wykonywania obliczeń decydują nawiasy oraz to, jak wysoki priorytet ma dane działanie. Nie jest to jednak jedyny sposób przedstawiania wyrażeń algebraicznych. Istnieją notacje, w których nie używa się nawiasów, a mimo to można właściwie określić kolejność wykonywania działań. Jedną z nich jest odwrotna notacja polska.

Cele lekcji

- Poznasz sposoby zapisu wyrażeń algebraicznych bez użycia nawiasów, w tym odwrotną notację polską.
- Dowiesz się, czym są dynamiczne struktury danych.
- Poznasz i zastosujesz dynamiczną strukturę danych o nazwie stos.
- Zamienisz wyrażenie algebraiczne zapisane w notacji tradycyjnej na zapis w odwrotnej notacji polskiej.
- Wykorzystasz odwrotną notację polską do obliczenia wartości wyrażenia arytmetycznego.

1.1. Sposoby zapisu wyrażeń algebraicznych

Notacja infiksowa

Stosowany powszechnie zapis wyrażeń algebraicznych, w którym znak działania występuje pomiędzy argumentami, to **notacja infiksowa**. Na przykład sumę a i b w tej notacji zapisujemy: $a + b$.

Notacja prefiksowa (notacja polska)

Innym sposobem zapisywania wyrażeń algebraicznych jest **notacja prefiksowa**, zwana też **notacją polską**. W jej przypadku znak działania stawia się przed argumentami, których dotyczy. Suma a i b będzie wyglądała następująco: $+ a b$. Znak działania można też postawić za argumentami: $a b +$. Ten sposób zapisu nazywa się

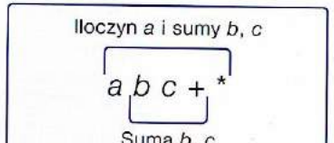
Notacja sufiksowa (odwrotna notacja polska – ONP)

notacją sufiksową lub **odwrotną notacją polską** (**ONP**). W notacjach prefiksowej i sufiksowej nie używa się nawiasów, a mimo to można określić kolejność wykonywania działań zgodnie z ich priorytetami.

Sposób interpretacji wyrażenia algebraicznego zapisanego w odwrotnej notacji polskiej (notacji sufiksowej) ilustruje rysunek 1.1.

Warto wiedzieć

Notację polską wymyślił polski matematyk Jan Łukasiewicz. Odwrotna notacja polska została zaproponowana przez australijskiego naukowca C.L. Hamblina i nazwana odwrotną notacją polską jako wyraz uznania dla osiągnięć Łukasiewicza.



Rys. 1.1. Wyrażenie algebraiczne $a * (b + c)$ przedstawione w odwrotnej notacji polskiej (notacji sufiksowej)

Na wyrażenia zapisane w odwrotnej notacji polskiej należy patrzeć od prawej do lewej. Wyrażenie algebraiczne $a\ b\ c\ +\ *$ z rysunku 1.1 oznacza iloczyn, którego pierwszym czynnikiem jest a, natomiast drugim suma b i c ($b\ c\ +$ oznacza sumę b i c, ponieważ znak + znajduje się bezpośrednio za b i c). Zatem zapis $a\ b\ c\ +\ *$ w odwrotnej notacji polskiej odpowiada wyrażeniu $a * (b + c)$ w notacji tradycyjnej.

Tabela 1.1 przedstawia przykłady wyrażeń algebraicznych w notacji tradycyjnej i odwrotnej notacji polskiej.

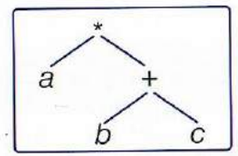
Warto wiedzieć

Odwrotna notacja polska była stosowana w niektórych kalkulatorach.

Notacja tradycyjna (infiksowa)	Odwrotna notacja polska (notacja sufiksowa)
$a + b * c$	$a\ b\ c\ * +$
$a * b + c$	$a\ b\ *\ c +$
$(a - b) * c$	$a\ b - c\ *$
$a / (b + c)$	$a\ b\ c + /$

Tabela 1.1. Przykłady wyrażeń algebraicznych zapisanych w notacji tradycyjnej i odwrotnej notacji polskiej

Na podstawie zapisu w odwrotnej notacji polskiej można łatwo przedstawić wyrażenie w postaci graficznej, jak na rysunku 1.2. Taki sposób zapisu nazywamy **drzewem wyrażenia algebraicznego**.



Rys. 1.2. Drzewo wyrażenia algebraicznego $a\ b\ c\ +\ *$

Drzewo wyrażenia algebraicznego

Ćwiczenie 1

Zapisz wyrażenie $(a + b) \cdot (c + d)$ w odwrotnej notacji polskiej oraz utwórz drzewo tego wyrażenia algebraicznego.

1.2. Czym jest stos?

W algorytmie zamieniającym wyrażenia z notacji tradycyjnej na ONP wykorzystamy strukturę danych o nazwie **stos** (ang. *stack*). Można ją porównać do stosu książek (rys. 1.3). Kolejną książkę dokłada się tylko na wierzch stosu i zdjąć można jedynie książkę leżącą na wierzchu. Podobnie w strukturze danych ostatnio dodany element jest wierzchołkiem stosu, a usunąć można tylko element będący wierzchołkiem. Na stosie można przechowywać wyłącznie dane tego samego typu. Jest on strukturą danych, z której pobieramy elementy w kolejności odwrotnej do ich wstawiania. Taką strategię dostępu nazywa się **LIFO** (od ang. *last in, first out* – ostatni wchodzi, pierwszy wychodzi).

Stos

LIFO

Rys. 1.3. Stos książek

Operacje na stosie Możemy wyróżnić następujące **operacje na stosie**:

- *push* – umieszczenie elementu na stosie,
- *pop* – usunięcie elementu będącego wierzchołkiem stosu,
- *top* – odczytanie wartości elementu będącego wierzchołkiem stosu,
- *empty* – sprawdzenie, czy stos jest pusty.

Dynamiczna struktura danych Stos jest przykładem **dynamicznej struktury danych**, czyli takiej, która może zmieniać swój rozmiar podczas działania programu. Liczbę elementów pamiętanych w dynamicznej strukturze danych można zwiększać lub zmniejszać w zależności od potrzeb.

Dobra rada

Jeśli nie wiesz, jak wymawiać obcojęzyczne nazwy umieszczone w podręczniku, skorzystaj z translatora, który daje możliwość odsłuchania tekstu, np. Tłumacza Google lub Bing Microsoft Translator.

Zapamiętaj

Stos jest dynamiczną strukturą danych, czyli taką, której rozmiar można zmieniać w trakcie działania programu. Operacje wykonuje się tylko na jednym końcu stosu. Do operacji tych należą: umieszczenie elementu na stosie (*push*), zdjęcie elementu ze stosu (*pop*), odczytanie wartości elementu będącego wierzchołkiem stosu (*top*) i sprawdzenie, czy stos jest pusty (*empty*).

1.3. Algorytm zamiany wyrażenia z notacji tradycyjnej na ONP

W algorytmie zamiany wyrażenia algebraicznego zapisanego w notacji tradycyjnej na zapis w odwrotnej notacji polskiej ograniczymy się do wyrażeń, w których mogą występować tylko:

- podstawowe działania algebraiczne dwuargumentowe – dodawanie, odejmowanie, mnożenie i dzielenie,
- dowolnie zagnieżdżone nawiasy okrągłe,
- jednoznakowe argumenty (litery, liczby jednocyfrowe).

Zakładamy także, że wyrażenie w notacji tradycyjnej jest zapisane bez błędów. Oto specyfikacja problemu:

Specyfikacja

Dane: w – napis reprezentujący wyrażenie algebraiczne w notacji tradycyjnej, w którym mogą występować znaki +, –, *, / jako znaki działań dwuargumentowych, nawiasy okrągłe oraz litery i liczby jednocyfrowe jako argumenty jednoznakowe.

Wynik: onp – napis reprezentujący wyrażenie w zapisane w odwrotnej notacji polskiej.

Algorytm zamiany będzie polegał na przeglądaniu wyrażenia zapisanego w notacji tradycyjnej od lewej do prawej i w zależności od rodzaju napotkanego znaku – podejmowaniu odpowiedniej decyzji.

Niektóre znaki podczas analizy wyrażenia wejściowego trzeba przechować przez pewien czas. Dotyczy to znaków działań (do czasu rozpatrzenia drugiego argumentu) oraz nawiasu otwierającego (do czasu napotkania w wyrażeniu odpowiadającego mu nawiasu zamykającego). Dane te przechowamy na stosie. Sposób postępowania ilustrują tabele 1.2 i 1.3 (s. 14) na przykładzie wyrażeń $a + b * c$ oraz $a - (b + c) * d + e$. W ostatniej kolumnie każdej tabeli elementy stosu zapisane są w kolejności od dna stosu do wierzchołka stosu (wierzchołkiem stosu jest pierwszy znak od prawej).

Tabela 1.2 przedstawia kolejne kroki prowadzące do zamiany wyrażenia $a + b * c$ na zapis w ONP. W kroku 4 znak mnożenia jest wkładany na stos po sprawdzeniu, jaki znak jest wierzchołkiem stosu. Jest to znak dodawania, a więc działania o priorytecie niższym od mnożenia. Gdyby było to działanie o priorytecie wyższym lub równym, przed włożeniem na stos należałoby znak takiego działania zdjąć ze stosu i dopisać do wyrażenia wynikowego. Po przeanalizowaniu wszystkich znaków wyrażenia wejściowego należy znaki działań pozostałe na stosie dopisać do wyrażenia wynikowego (kroki 6 i 7).

Krok	Rozpatrywany znak	Operacja/operacje	Wyrażenie w ONP	Elementy na stosie
1.	*a*	Dopisanie do ONP	*a*	
2.	+	Włożenie na stos	*a*	+
3.	*b*	Dopisanie do ONP	*a b*	+
4.	*	Włożenie na stos	*a b*	+ *
5.	*c*	Dopisanie do ONP	*a b c*	+ *
6.		Zdjęcie ze stosu i dopisanie do ONP znaku *	*a b c* *	+
7.		Zdjęcie ze stosu i dopisanie do ONP znaku +	*a b c* * +	

Tabela 1.2. Kolejne kroki podczas zamiany wyrażenia $a + b * c$ na zapis w ONP

Warto wiedzieć

Drzewo wyrażenia algebraicznego łatwiej jest utworzyć na podstawie zapisu wyrażenia w ONP niż na podstawie zapisu w notacji tradycyjnej.

Tabela 1.3 na s. 14 przedstawia kolejne kroki prowadzące do zamiany wyrażenia $a - (b + c) * d + e$ na zapis w ONP. W kroku 7 ze stosu zdejmujemy znak działania, który znajduje się nad nawiasem otwierającym. Znak ten dopisujemy do wyrażenia wynikowego, następnie zdejmujemy ze stosu nawias otwierający i go gubimy. W ten sposób zapiszemy w odwrotnej notacji polskiej fragment wyrażenia ujęty w nawiasy. W kroku 10, po napotkaniu w wyrażeniu znaku +, ze stosu zdejmujemy dwa znaki – jeden o priorytecie wyższym niż napotkany znak, drugi o priorytecie takim samym co napotkany znak – i dopisujemy je do wyrażenia wynikowego. Następnie znak + wkładamy na stos.

Ponieważ w wyrażeniu wejściowym nie ma już znaków do rozpatrzenia, zdejmujemy znak + ze stosu i dopisujemy go do wyrażenia wynikowego (krok 12).

Krok	Rozpatrywany znak	Operacja/operacje	Wyrażenie w ONP	Elementy na stosie
1.	*a*	Dopisanie do ONP	*a*	
2.	–	Włożenie na stos	*a*	–
3.	(	Włożenie na stos	*a*	–(
4.	*b*	Dopisanie do ONP	*a b*	–(
5.	+	Włożenie na stos	*a b*	–(+
6.	*c*	Dopisanie do ONP	*a b c*	–(+
7.	)	Zdjęcie ze stosu i dopisanie do ONP znaku + Zdjęcie ze stosu znaku (	*a b c* +	–
8.	*	Włożenie na stos	*a b c* +	–*
9.	*d*	Dopisanie do ONP	*a b c* + *d*	–*
10.	+	Zdjęcie ze stosu i dopisanie do ONP znaku * Zdjęcie ze stosu i dopisanie do ONP znaku – Włożenie na stos znaku +	*a b c* + *d* * –	+
11.	*e*	Dopisanie do ONP	*a b c* + *d* * – *e*	+
12.		Zdjęcie ze stosu i dopisanie do ONP znaku +	*a b c* + *d* * – *e* +	

Tabela 1.3. Kolejne kroki podczas zamiany wyrażenia *a* – (*b* + *c*) * *d* + *e* na zapis w ONP

Tabela 1.4 zawiera zestawienie operacji wykonywanych w algorytmie zamiany wyrażenia algebraicznego z notacji tradycyjnej na odwrotną notację polską w zależności od napotkanego znaku.

Rozpatrywany znak	Operacja/operacje
Litera/liczba jednocyfrowa	Dopisanie litery/liczby jednocyfrowej do wyniku
(	Włożenie nawiasu na stos
)	Zdjęcie ze stosu znaków działań do napotkania na stosie nawiasu otwierającego i dopisanie tych znaków działań do wyniku Zdjęcie ze stosu nawiasu otwierającego
Znak działania	Zdjęcie ze stosu znaków działań o priorytecie wyższym lub równym priorytetowi napotkanego znaku działania i dopisanie tych znaków działań do wyniku (jeśli takie znaki są) Włożenie rozpatrywanego znaku działania na stos
Koniec wyrażenia	Zdjęcie ze stosu i dopisanie do wyniku znaków działań, które pozostały na stosie

Tabela 1.4. Operacje w algorytmie zamiany wyrażenia algebraicznego z postaci tradycyjnej na zapis w ONP

Ćwiczenie 2

Zapisz kolejne kroki zamiany podanego wyrażenia na zapis w ONP.

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

Algorytm zamiany wyrażenia algebraicznego z notacji tradycyjnej na odwrotną notację polską, zgodny ze specyfikacją podaną na s. 12, można zapisać w pseudokodzie następująco:

```
onp ← ""
dla i ← 0, ..., długość w – 1 wykonuj
    jeśli w[i] = '(' to
        push(w[i])
        przejdź do następnego powtórzenia pętli
    jeśli w[i] = ')' to
        dopóki top() ≠ '(' wykonuj
            onp ← onp + top()
            pop()
        pop() // zdjęcie ze stosu znaku (
        przejdź do następnego powtórzenia pętli
    jeśli w[i] = '+' lub w[i] = '–' to
        dopóki nie empty() oraz top() ≠ '(' wykonuj
            onp ← onp + top()
            pop()
        push(w[i])
        przejdź do następnego powtórzenia pętli
    jeśli w[i] = '*' lub w[i] =  '/' to
        jeśli nie empty() oraz
              (top() = '*' lub top() = '/') to
            onp ← onp + top()
            pop()
        push(w[i])
    w przeciwnym przypadku onp ← onp + w[i]
dopóki nie empty() wykonuj
    onp ← onp + top()
    pop()
```

Zakładamy, że mamy do dyspozycji **stos** oraz **operacje na stosie**. Zgodnie ze specyfikacją w wyrażeniu mogą wystąpić tylko cztery działania: dwa o niższym priorytecie i dwa o wyższym. Napotkanie dodawania lub odejmowania powoduje zdjęcie ze stosu i dopisanie do wyniku działań znajdujących się na stosie aż do napotkania ewentualnego nawiasu otwierającego (ponieważ nie ma działań o niższym priorytecie). Natomiast napotkanie mnożenia lub dzielenia powoduje zdjęcie ze stosu i dopisanie do wyniku tylko innego mnożenia lub dzielenia (ponieważ nie ma działań o wyższym priorytecie).

Stos, s. 11
Operacje na stosie, s. 12

Jeśli napotkany znak nie jest znakiem działania ani nawiasem, to musi być argumentem. Dopisanie argumentu do wyniku realizuje instrukcja podana jako przeciwny przypadek do ostatniej instrukcji **jeśli**.

Ćwiczenie 3

Zamień poniższe wyrażenia na zapis w odwrotnej notacji polskiej zgodnie z omówionym algorytmem zapisanym w pseudokodzie.

a. $((a+b)\cdot c-d)\cdot e+f\cdot g$

b. $\dfrac{a\cdot(b+c)-d\cdot(e+f)}{g+h}$

1.4. Implementacja algorytmu zamiany wyrażenia z notacji tradycyjnej na ONP

Zaimplementujemy teraz w języku C++ omówiony algorytm zamiany wyrażenia algebraicznego z notacji tradycyjnej na odwrotną notację polską. Wykorzystamy w tym celu szablon stosu z **biblioteki STL** (ang. *Standard Template Library* – standardowa biblioteka szablonów). Aby móc skorzystać z szablonu, na początku programu należy dołączyć bibliotekę `stack` – dyrektywa `#include <stack>`. Ogólna **deklaracja stosu** jest następująca:

Biblioteka STL

Deklaracja stosu

```
stack<typ elementów stosu> nazwa_stosu;
```

W naszym programie elementami przechowywanymi na stosie będą znaki, a więc deklaracja zmiennej typu **stack** o nazwie stos będzie miała postać:

```
stack<char> stos;
```

Typ **stack** jest typem obiektowym, zatem udostępnia metody do przetwarzania danych. Należą do nich m.in. **metody** push, pop, top, **empty**. Ich działanie odpowiada wymienionym w tym temacie **operacjom na stosie**. Parametrem metody push jest element, który chcemy wstawić na stos. Metody top, pop, **empty** są bezparametrowe. Korzystając z metody, stosujemy notację z kropką – po nazwie stosu zapisujemy kropkę i nazwę metody, np. `stos.top()`.

Metody push, pop, top, empty dla klasy stack

Operacje na stosie, s. 12

Odwołanie do elementu stosu, gdy stos jest pusty, jest błędem. Żeby przy odwołaniu do wierzchołka stosu nie sprawdzać, czy stos jest pusty, wprowadzimy **wartownika**. Będzie on pełnił funkcję dna stosu. Za wartownika przyjmiemy znak, który nie może wystąpić w wyrażeniu wejściowym (np. znak #).

Wartownik to element ułatwiający odnalezienie końca danych.

Kod źródłowy funkcji realizującej algorytm zamiany wyrażenia algebraicznego z postaci tradycyjnej na odwrotną notację polską, zapisany wcześniej w pseudokodzie, może wyglądać następująco.


```
string ONP(string w)
{
    stack<char> stos;
    stos.push('#');
    string onp="";
    for (int i=0;i<w.size();i++)
    {
        if (w[i]=='(')
        {
            stos.push('(');
            continue;
        }
        if (w[i]==')')
        {
            while (stos.top()!='(')
            {
                onp=onp+stos.top();
                stos.pop();
            }
            stos.pop();
            continue;
        }
        if (w[i]=='+' || w[i]=='-')
        {
            while (stos.top()!='#' && stos.top()!='(')
            {
                onp=onp+stos.top();
                stos.pop();
            }
            stos.push(w[i]);
            continue;
        }
        if (w[i]=='*' || w[i]=='/')
        {
            if (stos.top()=='*' || stos.top()=='/')
            {
                onp=onp+stos.top();
                stos.pop();
            }
            stos.push(w[i]);
        }
        else onp=onp+w[i];
    }
    while (stos.top()!='#')
    {
        onp=onp+stos.top(); stos.pop();
    }
    stos.pop();
    return onp;
}
```

Kod źródłowy funkcji zamieniającej wyrażenie algebraiczne z postaci tradycyjnej na odwrotną notację polską (z wielokrotnym wykorzystaniem instrukcji warunkowej)

Dobra rada

Kiedy korzystasz z typu **stack**, pamiętaj o dołączeniu do programu biblioteki `stack`.

W linii 3 została zadeklarowana zmienna stos, a w linii 4 wstawiono na stos znak # jako dno stosu (wartownik). W linii 5 zadeklarowana jest zmienna onp i nadana jest jej wartość – napis pusty. W liniach 6–43 znajduje się główna pętla przeglądająca kolejne znaki wyrażenia algebraicznego i w zależności od rozpatrywanego znaku wykonująca operacje zgodnie z omówionym algorytmem. Po rozpoznaniu znaku i wykonaniu odpowiadających mu czynności należy przejść do analizy kolejnego znaku, pomijając pozostałe instrukcje w pętli. Aby to zrobić, wykorzystaliśmy **instrukcję continue**, która powoduje pominięcie instrukcji do końca pętli i przejście do następnego powtórzenia pętli. Pętla w liniach 44–47 zdejmuje ze stosu i dopisuje do wyniku znaki działań, które zostały na stosie po zakończeniu analizy wyrażenia algebraicznego. W linii 48 ze stosu usuwany jest wartownik (stos będzie pusty).

Instrukcja continue

Warto wiedzieć

Zamiast korzystać z instrukcji **continue**, można użyć instrukcji:
if ...
else if ...

Rysunek 1.4 przedstawia przykład wywołania programu wykorzystującego funkcję ONP, przedstawioną na s. 17, i zamieniającego zapis wyrażenia z postaci tradycyjnej na odwrotną notację polską.

```
Podaj wyrazenie: (a+b)/c-(b+d)
ONP: ab+c/bd+-
```

Rys. 1.4. Przykład wywołania programu zamieniającego wyrażenie zapisane w postaci tradycyjnej na zapis w ONP

Ćwiczenie 4

Napisz program zamieniający wyrażenie zapisane w notacji tradycyjnej na zapis w odwrotnej notacji polskiej zgodnie ze specyfikacją podaną na s. 12.

Instrukcja wyboru switch

W głównej pętli funkcji ONP (pętli **for**) zamiast wielokrotnie używać instrukcji warunkowej, można skorzystać z **instrukcji wyboru switch**. Składnia instrukcji **switch** jest następująca:

```
switch (zmienna)
{
    case wartość_1:
        // instrukcje, gdy zmienna==wartość_1
        break;
    case wartość_2:
        // instrukcje, gdy zmienna==wartość_2
        break;
    ...
    case wartość_n:
        // instrukcje, gdy zmienna==wartość_n
        break;
    default:
        // instrukcje, gdy zmienna nie przyjmuje
        // żadnej z wymienionych wcześniej wartości
}
```


Aby skorzystać z instrukcji **switch**, wartości, jakie może przyjąć zmienna, muszą być typu całkowitoliczbowego. W zależności od wartości zmiennej wykonywana jest jedna z części kodu zaczynających się od słowa kluczowego **case**. Na przykład gdy zmienna przyjmie wartość wartość_2, program wykona instrukcje od części rozpoczynającej się od **case** wartość_2. Część **default** jest opcjonalna (może nie wystąpić) – podobnie jak część **else** w instrukcji warunkowej **if**. Dzięki **instrukcji break** można przerwać wykonywanie kolejnych instrukcji zapisanych w **switch**. Gdyby zabrakło instrukcji **break**, po wykonaniu instrukcji dla danej wartości zmiennej program przeszedłby do kolejnych linii w instrukcji **switch**. Zauważ, że jeśli nie wystąpi część **default**, nie ma potrzeby używania instrukcji **break** w części z ostatnim **case**. Instrukcja **break** ma także zastosowanie w pętlach – dzięki niej można zatrzymać wykonywanie instrukcji w pętli i wyjść z pętli.

Instrukcja break

Fragment kodu źródłowego funkcji ONP z wykorzystaniem instrukcji **switch** (zamiast linii 7–43) może wyglądać tak jak poniżej. Zauważ, że zmienną jest znak, któremu w języku C++ odpowiada kod ASCII. Zmienna przyjmuje zatem wartości, które są liczbami całkowitymi.

```
1.  switch (w[i])
2.  {
3.      case '(':
4.          stos.push('('); break;
5.      case ')':
6.          while (stos.top()!='(')
7.          {
8.              onp=onp+stos.top(); stos.pop();
9.          }
10.         stos.pop(); break;
11.     case '+':
12.         while (stos.top()!='#' && stos.top()!='(')
13.         {
14.             onp=onp+stos.top(); stos.pop();
15.         }
16.         stos.push('+'); break;
17.     case '-':
18.         while (stos.top()!='#' && stos.top()!='(')
19.         {
20.             onp=onp+stos.top(); stos.pop();
21.         }
22.         stos.push('-'); break;
23.     case '*':
24.         if (stos.top()=='*' || stos.top()=='/')
25.         {
26.             onp=onp+stos.top(); stos.pop();
27.         }
28.         stos.push('*'); break;
```

Fragment kodu źródłowego funkcji zamieniającej wyrażenie algebraiczne z postaci tradycyjnej na odwrotną notację polską (z wykorzystaniem instrukcji switch)

Warto wiedzieć

Stos w programie zamieniającym wyrażenie z notacji tradycyjnej na ONP można też zaimplementować, wykorzystując do tego zmienną typu `string`.

```
29.     case '/':
30.         if (stos.top()=='*' || stos.top()=='/')
31.         {
32.             onp=onp+stos.top(); stos.pop();
33.         }
34.         stos.push('/'); break;
35.     default:
36.         onp=onp+w[i];
37. }
```

Ćwiczenie 5

Napisz program zamieniający wyrażenie zapisane w notacji tradycyjnej na zapis w odwrotnej notacji polskiej z wykorzystaniem instrukcji wyboru **switch**.

Zapamiętaj

Do zamiany wyrażenia z notacji tradycyjnej na odwrotną notację polską wykorzystuje się strukturę danych o nazwie stos. Aby wybrać jeden przypadek z wielu, można wielokrotnie zastosować instrukcję warunkową `if` lub skorzystać z instrukcji wyboru `switch`.

1.5. Obliczanie wartości wyrażenia arytmetycznego zapisanego w ONP

Zajmiemy się teraz problemem obliczania wartości wyrażenia zapisanego w odwrotnej notacji polskiej zgodnie z poniższą specyfikacją.

Specyfikacja

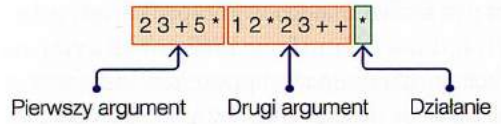
Dane: `onp` – napis reprezentujący wyrażenie arytmetyczne w odwrotnej notacji polskiej, w którym mogą występować znaki +, –, *, / jako znaki działań dwuargumentowych oraz liczby jednocyfrowe. Znak / oznacza wyznaczanie części całkowitej z dzielenia.

Wynik: `wart` – liczba całkowita będąca wartością wyrażenia onp.

Algorytm rekurencyjny

Rekurencja (rekursja), podręcznik *Informatyka na czasie 2. Zakres rozszerzony*, s. 236

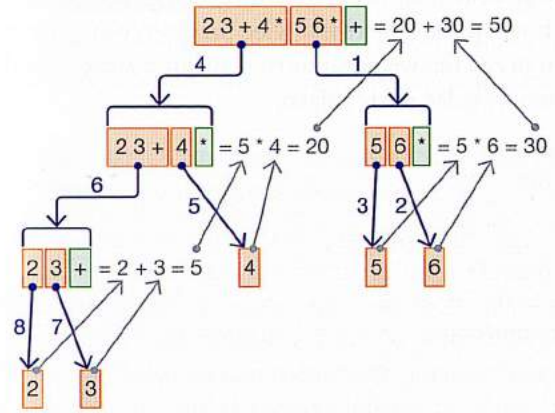
Najpierw **rekurencyjnie** obliczymy wartości argumentów działania, które trzeba wykonać jako ostatnie. W odwrotnej notacji polskiej znak tego działania jest ostatnim znakiem zapisu. Rysunek 1.5 przedstawia przykładowe wyrażenie z podziałem na fragmenty, których wartości trzeba wyznaczyć rekurencyjnie.



Rys. 1.5. Podział wyrażenia na argumenty wyznaczane rekurencyjnie

Jeśli w odwrotnej notacji polskiej ostatni znak wyrażenia jest cyfrą, to dane wyrażenie jest stałą. W takim przypadku zgodnie ze specyfikacją stałą tą jest liczba jednocyfrowa. Nie ma wówczas wywołania rekurencyjnego, a wynikiem jest dana liczba.

Na rysunku 1.6 są przedstawione wywołania rekurencyjne dla wyrażenia 2 3 + 4 * 5 6 * +. Liczby przy strzałkach oznaczają kolejność tych wywołań. Zielony prostokąt wskazuje ostatni znak badanego wyrażenia, a pomarańczowe prostokąty – argumenty, które zostaną obliczone rekurencyjnie. Szarym kolorem oznaczone są zwracane wyniki.



Rys. 1.6. Schemat rekurencyjnego obliczania wartości wyrażenia zapisanego w ONP

Oto pseudokod funkcji realizującej algorytm rekurencyjny:

```
funkcja Oblicz(onp)
    znak ← ostatni znak wyrażenia w onp
    onp ← onp bez ostatniego znaku
    jeśli znak jest cyfrą to
        zwróć wartość(znak) i zakończ
    w przeciwnym przypadku
        arg2 ← Oblicz(onp)
        arg1 ← Oblicz(onp)
        jeśli znak = '+' to
            zwróć arg1 + arg2 i zakończ
        jeśli znak = '–' to
            zwróć arg1 – arg2 i zakończ
        jeśli znak = '*' to
            zwróć arg1 * arg2 i zakończ
        jeśli znak = '/' to
            zwróć arg1 div arg2 i zakończ
```

Warto wiedzieć

Funkcja `Oblicz` musi modyfikować zmienną `onp`, aby w kolejnych wywołaniach rekurencyjnych parametrem tej funkcji było odpowiednio skrócone wyrażenie.

Wartością funkcji Oblicz jest liczba. Dlatego w przypadku, gdy ostatni znak wyrażenia jest cyfrą, dokonujemy konwersji znaku cyfry na liczbę jednocyfrową. Operacja ta oznaczona jest przez wartość(znak).

Dla wyrażenia przedstawionego na rysunku 1.6 ze s. 21 nastąpią poniższe wywołania funkcji Oblicz:

```
Oblicz("23+4*56*+") = 50
    Oblicz("23+4*56*") = 30
        Oblicz("23+4*56") = 6, warunek początkowy
        Oblicz("23+4*5") = 5, warunek początkowy
    Oblicz("23+4*") = 20
        Oblicz("23+4") = 4, warunek początkowy
        Oblicz("23+") = 5
            Oblicz("23") = 3, warunek początkowy
            Oblicz("2") = 2, warunek początkowy
```

Zapisując funkcję Oblicz w języku C++, należy pamiętać, że zmiany dokonane na parametrze onp (usunięcie ostatniego znaku z wyrażenia) muszą zostać zachowane. Parametr jest więc przekazywany przez referencję. W tym celu przed nazwą parametru stawiamy znak &. Kod źródłowy funkcji może wyglądać następująco:

Kod źródłowy funkcji obliczającej wartość wyrażenia arytmetycznego zapisanego w ONP, realizującej algorytm rekurencyjny

```
1.  int Oblicz(string &onp)
2.  {
3.      char znak=onp[onp.size()-1];
4.      onp.erase(onp.size()-1,1);
5.      if (znak>='0' && znak<='9')
6.          return znak-48;
7.      else
8.      {
9.          int arg2=Oblicz(onp);
10.         int arg1=Oblicz(onp);
11.         switch (znak)
12.         {
13.             case '+':
14.                 return arg1+arg2;
15.             case '-':
16.                 return arg1-arg2;
17.             case '*':
18.                 return arg1*arg2;
19.             case '/':
20.                 return arg1/arg2;
21.         }
22.     }
23. }
```

Dobra rada

Znaki parametru onp są usuwane z wyrażenia przez funkcję rekurencyjną Oblicz. Jeśli po jej wykonaniu potrzebny byłby jeszcze zapis wyrażenia, zapamiętaj go w zmiennej pomocniczej przed wywołaniem funkcji Oblicz.

W linii 3 zmienna znak przyjmuje wartość ostatniego znaku wyrażenia onp będącego parametrem, a w linii 4 ten znak jest usuwany z wyrażenia. Jeśli jest on cyfrą, to funkcja Oblicz zwróci liczbę jednocyfrową.

Aby zwrócić tę liczbę, od kodu ASCII cyfry odejmowana jest wartość 48, czyli kod ASCII znaku '0' (linia 6). W liniach 9 i 10 obliczane są rekurencyjnie argumenty działania (najpierw prawy argument). Następnie w instrukcji **switch** obliczany jest wynik funkcji (linie 11–21). Instrukcja **return** kończy działanie funkcji. Ponieważ została użyta w instrukcji **switch**, nie jest już potrzebna instrukcja **break**.

Rysunek 1.7 przedstawia wywołanie programu obliczającego wartość wyrażenia arytmetycznego podanego w odwrotnej notacji polskiej.

Dobra rada

Zapis **return** znak-48 w linii 6 możesz zastąpić instrukcją: **return** znak-'0'

```
Podaj wyrazenie w ONP: 65*42-1+/
Wartosc wyrazenia: 10
```

Rys. 1.7. Przykład wywołania programu obliczającego wartość wyrażenia arytmetycznego podanego w odwrotnej notacji polskiej

Ćwiczenie 6

Napisz program obliczający rekurencyjnie wartość wyrażenia zapisanego w ONP zgodnie ze specyfikacją podaną na s. 20.

Algorytm z wykorzystaniem stosu

Będziemy teraz analizować wyrażenie od lewej do prawej. Ponieważ w zapisie ONP znak działania znajduje się za argumentami, których dotyczy działanie, argumenty musimy zapamiętać do momentu, aż odczytamy znak działania. Dlatego przechowamy je na stosie. Działanie dotyczy dwóch argumentów znajdujących się na górze stosu (prawy argument działania to wierzchołek stosu, a lewy to element znajdujący się bezpośrednio pod nim). Należy więc zdjąć dwa elementy ze stosu, wykonać działanie arytmetyczne i wynik ponownie włożyć na stos. Po przejrzeniu całego wyrażenia na stosie pozostanie jeden element – wynik działania. Elementami stosu będą liczby całkowite.

Warto wiedzieć

W omówionym algorytmie rekurencyjnym, znajdującym wartość wyrażenia arytmetycznego zapisanego w ONP, wyrażenie było przeglądane od prawej do lewej. Jest to ogólna cecha algorytmów rekurencyjnych – przeglądają one dane w odwrotnej kolejności niż algorytmy iteracyjne.

A to ciekawe

Stos w translatorach języków programowania

Twórcą koncepcji stosu jako struktury danych jest niemiecki matematyk Friedrich Bauer (na zdjęciu). Przedstawił ją w roku 1955. Stos jest wykorzystywany m.in. w translatorach języków programowania do pamiętania zmiennych lokalnych i parametrów funkcji. Właśnie z tego powodu przy obliczaniu złożoności pamięciowej algorytmów rekurencyjnych dodaje się składnik odpowiedzialny za rozmiar stosu. Friedrich Bauer jest też autorem popularnego terminu „inżynieria oprogramowania".

Tabela 1.5 przedstawia kolejne kroki podczas wyznaczania wartości wyrażenia 2 3 * 4 5 * + z wykorzystaniem stosu.

Krok	Rozpatrywany znak	Operacja/operacje	Elementy na stosie
1.	2	Włożenie na stos	2
2.	3	Włożenie na stos	2 3
3.	*	Zdjęcie ze stosu 3 Zdjęcie ze stosu 2 Włożenie na stos iloczynu 2 i 3	6
4.	4	Włożenie na stos	6 4
5.	5	Włożenie na stos	6 4 5
6.	*	Zdjęcie ze stosu 5 Zdjęcie ze stosu 4 Włożenie na stos iloczynu 4 i 5	6 20
7.	+	Zdjęcie ze stosu 20 Zdjęcie ze stosu 6 Włożenie na stos sumy 6 i 20	26
8.		Zdjęcie ze stosu wyniku wyrażenia	

Tabela 1.5. Kolejne kroki podczas obliczania wartości wyrażenia 2 3 * 4 5 * + z wykorzystaniem stosu

Algorytm obliczający wartość wyrażenia arytmetycznego z wykorzystaniem stosu zgodnie ze specyfikacją podaną na s. 20 można zapisać w pseudokodzie następująco:

```
dla i ← 0, ..., długość onp – 1 wykonuj
    jeśli onp[i] jest cyfrą to push(wartość(onp[i]))
    w przeciwnym przypadku
        arg2 ← top()
        pop()
        arg1 ← top()
        pop()
        jeśli znak = '+' to
            push(arg1 + arg2)
            przejdź do następnego powtórzenia pętli
        jeśli znak = '-' to
            push(arg1 - arg2)
            przejdź do następnego powtórzenia pętli
        jeśli znak = '*' to
            push(arg1 * arg2)
            przejdź do następnego powtórzenia pętli
        jeśli znak = '/' to
            push(arg1 div arg2)
wart ← top()
pop()
```

Warto wiedzieć

Wierzchołkiem stosu jest prawy argument działania. Kolejność zdejmowania argumentów ze stosu ma znaczenie dla działań, które nie są przemienne.

Oto kod źródłowy funkcji realizującej podany algorytm:

```
1.  int Oblicz(string onp)
2.  {
3.      stack<int> stos;
4.      for(int i=0;i<onp.size();i++)
5.          if (onp[i]>='0' && onp[i]<='9')
6.              stos.push(onp[i]-48);
7.          else
8.          {
9.              int arg2=stos.top(); stos.pop();
10.             int arg1=stos.top(); stos.pop();
11.             switch (onp[i])
12.             {
13.                 case '+':
14.                     stos.push(arg1+arg2);
15.                     break;
16.                 case '-':
17.                     stos.push(arg1-arg2);
18.                     break;
19.                 case '*':
20.                     stos.push(arg1*arg2);
21.                     break;
22.                 case '/':
23.                     stos.push(arg1/arg2);
24.             }
25.         }
26.     int wart=stos.top();
27.     stos.pop();
28.     return wart;
29. }
```

Kod źródłowy funkcji obliczającej wartość wyrażenia arytmetycznego zapisanego w ONP, wykorzystującej stos

W linii 3 deklarowany jest stos, którego elementami będą liczby całkowite (zmienna `stos` typu **`stack`** o elementach typu **`int`**). W pętli (linie 4–25) przeglądane są kolejne znaki wyrażenia od lewej do prawej. Jeśli znak jest cyfrą, to jest ona zapisywana na stosie jako liczba jednocyfrowa (linie 5–6). W przeciwnym przypadku ze stosu są zdejmowane dwie liczby, które są zapamiętywane odpowiednio w zmiennych `arg2` i `arg1`, czyli najpierw prawy argument działania, potem lewy (linie 9–10). W liniach 11–24 wkładany jest na stos wynik analizowanego działania. Po zakończeniu działania pętli na stosie znajduje się jedna liczba, która jest obliczoną wartością wyrażenia. Jest ona zdejmowana ze stosu i zapamiętywana w zmiennej `wart` (linie 26–27), której wartość jest wynikiem funkcji `Oblicz` (linia 28).

Ćwiczenie 7

Napisz program obliczający wartość wyrażenia zapisanego w ONP zgodnie ze specyfikacją podaną na s. 20. Wykorzystaj stos.

Podsumowanie

- Sposób zapisu wyrażeń algebraicznych, w którym znak działania występuje pomiędzy argumentami, nazywamy notacją infiksową.
- W notacji prefiksowej (notacji polskiej) znak działania występuje przed argumentami, natomiast w notacji sufiksowej (odwrotnej notacji polskiej) znak działania zapisuje się po argumentach.
- W notacji polskiej oraz odwrotnej notacji polskiej nie używa się nawiasów.
- Dynamiczne struktury danych charakteryzują się tym, że ich rozmiar (liczba elementów) może się zmieniać podczas działania programu.
- Stos to dynamiczna struktura danych typu LIFO (ang. *last in, first out*), w której operacje są wykonywane na jej jednym końcu. Przykładowe operacje na stosie to: *push* (umieszczenie elementu na stosie), *pop* (zdjęcie elementu ze stosu), *top* (odczytanie wartości elementu będącego wierzchołkiem stosu) i *empty* (sprawdzenie, czy stos jest pusty).
- Algorytm zamiany wyrażenia z notacji tradycyjnej na odwrotną notację polską wykorzystuje stos. Na stosie przechowywane są nawiasy otwierające oraz znaki działań.
- Algorytm obliczania wartości wyrażenia arytmetycznego zapisanego w odwrotnej notacji polskiej może wykorzystywać stos. Wówczas na stosie przechowywane są argumenty działań arytmetycznych.
- Aby wybrać jeden przypadek z wielu, można wielokrotnie użyć instrukcji warunkowej **`if`** lub instrukcji wyboru **`switch`**.

Zadania

* **1** Napisz program, który wczyta napis złożony wyłącznie z nawiasów okrągłych oraz sprawdzi, czy są one właściwie sparowane. Program powinien wypisać słowo „TAK" lub „NIE". Na przykład dla napisu „(())" program zwróci „TAK", a dla napisu „)(()" – „NIE".

* **2** Napisz program zamieniający wyrażenie algebraiczne zapisane w notacji tradycyjnej na wyrażenie w odwrotnej notacji polskiej. W wyrażeniu mogą wystąpić argumenty jednoznakowe (litery lub liczby jednocyfrowe), nawiasy okrągłe oraz dwuargumentowe działania arytmetyczne: dodawanie (+), odejmowanie (–), mnożenie (*), wyznaczanie części całkowitej z dzielenia (/) oraz reszty z dzielenia (%).

* **3** Napisz program obliczający wartość wyrażenia arytmetycznego zapisanego w odwrotnej notacji polskiej. Wyrażenie może się składać z cyfr jako argumentów jednoznakowych oraz z dwuargumentowych działań arytmetycznych: dodawania (+), odejmowania (–), mnożenia (*), wyznaczania części całkowitej z dzielenia (/) oraz reszty z dzielenia (%).

** **4** Napisz program, który wczyta liczbę całkowitą dodatnią, a następnie wypisze tworzące ją cyfry w kolejności od cyfry najbardziej znaczącej do cyfry jedności, każdą w oddzielnym wierszu. Wykorzystaj stos liczb całkowitych.

** **5** Napisz program, który wczyta napis złożony z maksymalnie trzech rodzajów nawiasów: (), [] oraz {}, a następnie sprawdzi, czy są one właściwie sparowane. Kolejność par nawiasów jest dowolna. Na przykład napis „({})" uznajemy za poprawny, a napis „({)}" – za niepoprawny. Program powinien wypisać słowo „TAK" lub „NIE".

** **6** Napisz program zamieniający wyrażenie algebraiczne zapisane w notacji tradycyjnej na wyrażenie zapisane w odwrotnej notacji polskiej. Wyrażenie może się składać z argumentów jednoznakowych (liter i liczb jednocyfrowych), nawiasów okrągłych oraz dwuargumentowych działań arytmetycznych: dodawania (+), odejmowania (–), mnożenia (*), wyznaczania części całkowitej z dzielenia (/) oraz potęgowania (^).

** **7** Napisz program wczytujący napis składający się z cyfr jako argumentów jednoznakowych oraz dwuargumentowych działań arytmetycznych: dodawania (+), odejmowania (–), mnożenia (*) oraz wyznaczania części całkowitej z dzielenia (/) i sprawdzający, czy jest to prawidłowe wyrażenie arytmetyczne zapisane w odwrotnej notacji polskiej. Program powinien wypisać słowo „TAK" lub „NIE".

*** **8** Napisz program, który wczyta napis złożony z maksymalnie trzech rodzajów nawiasów: (), [] oraz {}, a następnie sprawdzi, czy są one właściwie sparowane. Kolejność par nawiasów ma znaczenie i jest następująca (od najbardziej wewnętrznego): (), [], {}. Na przykład napis „{()}" jest poprawny, a napis „([])" nie jest. Program powinien wypisać słowo „TAK" lub „NIE".

*** **9** Zapisz w wybranej przez siebie notacji (lista kroków, schemat blokowy, pseudokod, język programowania) algorytm sortowania szybkiego bez wykorzystania rekurencji. Skorzystaj ze stosu.