

Tomasz "Zyx" Jędrzejewski

Wyrażenia w PHP

Wersja 2.0 (23 lipca 2008)



Szczegółowe informacje o licencji znajdują się pod artykułem.

www.zyxist.com

Wyrażenia w PHP

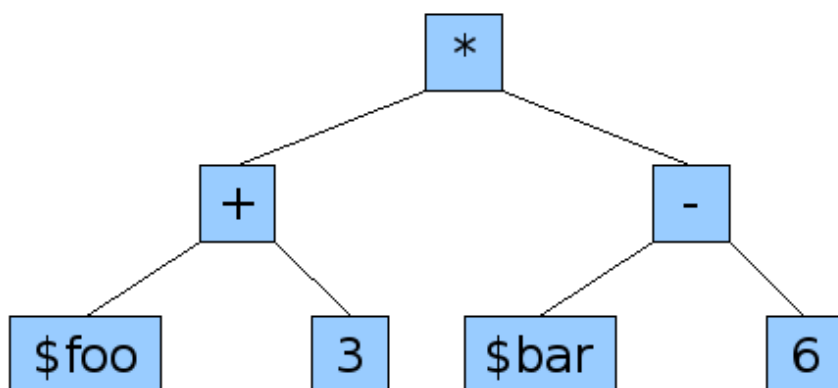
Czy wiesz, co ma wspólnego ze sobą zapis `$zmienna = 15` i `$licznik >= $maks` w instrukcji IF? A zapis `1` z `$zmienna`? Z pozoru nic, w praktyce wszystko - wszystkie podane przykłady są tym samym tworem - wyrażeniem. Jest to jedno z najważniejszych zagadnień programowania, lecz zwykle jest ono pomijane, lub omawiane "po macoszemu" w rozmaitych kursach i publikacjach. Artykuł ten pokazuje, jak wyjść poza schematy i poznać prawdziwą potęgę wyrażań, która znacznie ułatwia wiele zadań.

Zaczynamy

Na początek podamy definicję wyrażenia, z której wynikają niemal wszystkie jej własności. Wyrażeniem nazywamy:

1. Pojedynczą wartość (liczbę, ciąg tekstowy itd.)
2. Zmienną
3. Wywołanie funkcji lub metody
4. Inne wyrażenie lub grupę wyrażań połączonych operatorem, który wykonuje na nich działania i produkuje wynik.

Zasadniczą cechą wyrażenia jest to, że zawsze generuje ono jakiś wynik. W przypadku zwykłych wartości jest to oczywiście ta wartość, dla zmiennych – wartość zmiennych, dla funkcji – wynik ich działania. Zauważmy, że definicja wyrażenia jest rekurencyjna. Wynik mniejszego wyrażenia może być wartością wejściową dla innego, co można rozrysować w postaci pewnego rodzaju drzewa:



Wyrażenia mogą być umieszczane w wielu miejscach skryptu. Są argumentami struktur kontrolnych, występują również samodzielnie i to właśnie je standardowo kończymy średnikiem. Teoretycznie więc poniższy skrypt jest całkowicie poprawny, pomimo iż w zasadzie nic nie robi:

```
<?php
    10;
    $zmienna;
?>
```

Dokładnie na takiej zasadzie wywoływane są funkcje, z tym że one zazwyczaj już coś robią.

O wiele ciekawsze wyrażenia możemy budować przy pomocy operatorów, które pobierają wartości od jednego do trzech wyrażań i produkują w ich miejsce nową wartość. Przykładowo, operator sumowania `+` bierze liczby z lewej i z prawej strony, a następnie generuje ich sumę, która może z kolei być zastosowana

jako wartość wejściowa dla innego wyrażenia lub struktury kontrolnej:

```
<?php
    if($a + $b)
    {
        echo 'Zrob cos';
    }
?>
```

Istnieje także specjalna grupa operatorów, w których przynajmniej jeden argument nie może być dowolnym wyrażeniem. Wśród nich jest przypisanie, gdzie po lewej stronie bezwzględnie musi znaleźć się zmienna:

```
<?php
    $a = 5;      // dobrze
    5 = $a;     // źle
?>
```

Jednak bez względu na to, każdy operator *zawsze* jakąś wartość zwraca – w powyższym przypadku jest to przypisana do zmiennej wartość.

Operatory wykonywane są w określonej kolejności (podobnie jak mnożenie i dodawanie w matematyce), dzięki czemu zapis taki, jak poniżej, jest jednoznaczny:

```
<?php
    $wynik = 123 + 456;
?>
```

Wiedząc, że dodawanie ma wyższy priorytet, niż przypisanie, możemy przewidzieć, że PHP w pierwszej kolejności zsumuje 123 oraz 456 i dopiero ten wynik przypisze do zmiennej. W razie konieczności, kolejność zawsze można zmienić przy pomocy nawiasów:

```
<?php
    // dobrze:
    $wynik = (123 + 456) * 789;
    // też dobrze:
    ($wynik = 123) + 456;
?>
```

PHP dostrzega nawiasy, dlatego pierwsze wyrażenie nie będzie sumą 123 oraz iloczynu 456 razy 789, tylko iloczynem sumy 123 i 456 z 789. Poprawne jest także drugie wyrażenie. Podstawia ono 123 do zmiennej *\$wynik*, po czym sumuje tę liczbę z 456, lecz wynik tego obliczenia jest już gubiony, gdyż nie napisaliśmy, by trafił on gdzieś indziej. Nic nie stoi na przeszkodzie, aby jednak do jakiejś zmiennej go przypisać; wszak w dalszym ciągu jest to wyrażenie, które można umieścić po prawej stronie operatora przypisania:

```
<?php
    $wynik2 = ($wynik = 123) + 456;
?>
```

Po wykonaniu tego skryptu w zmiennej *\$wynik* będzie liczba 123, a w *\$wynik2* 579 (suma 123 i 456). Czy potrafisz powiedzieć, dlaczego?

Zabawy operatorem przypisania

Nie jest to jedyna rzecz, którą można zrobić z operatorem przypisania. Zastanów się, czy podany niżej przykład jest poprawny, a jeśli tak – jaki da wynik:

```
<?php
    $zmienna1 + $zmienna2 = $zmienna3 + $zmienna4;
```

```
?>
```

Chociaż napisaliśmy, że po lewej stronie operatora przypisania musi być zmienna, powyższy przykład jest poprawny. Zwróć uwagę, że zmienna faktycznie jest i nie przeszkadza to w niczym, aby wcześniej była część innego wyrażenia. Zgodnie z kolejnością wykonywania działań, na początku powinny być wykonane sumowania $\$zmienna1+\$zmienna2$ oraz $\$zmienna3+\$zmienna4$. Drugie z nich oczywiście się wykonuje. Jednak z powodu wymagań przypisania, należy zmodyfikować kolejność w pierwszym, by zachować poprawność. Dlatego parser wykona następujące kroki:

1. Zsumuje $\$zmienna3$ i $\$zmienna4$.
2. Przypisze wynik do $\$zmienna2$
3. Zsumuje $\$zmienna1$ oraz wynik przypisania, czyli de facto nową wartość zmiennej $\$zmienna2$.

Jak widać, operator przypisania, mimo pewnych ograniczeń, wciąż jest elastycznie obsługiwany przez parser. Wykorzystajmy to do szybkiego zainicjowania paru zmiennych tą samą wartością:

```
<?php
    $zmienna1 = $zmienna2 = $zmienna3 = $zmienna4 = 5;
?>
```

To wyrażenie wykonuje się od prawej do lewej. Najpierw parser przypisuje 5 do zmiennej $\$zmienna4$, produkuje wynik również równy 5, przypisuje go do zmiennej $\$zmienna3$ i tak dalej, aż wszystkie zmienne nie zostaną zainicjowane.

Flagi

Czasem zdarza się, że stan pewnego obiektu opisany jest szeregiem włączników TAK/NIE, które mówią, co jest aktywne, a co nie. Przy programowaniu obiektowym dość często zdarza się, że każdy włącznik posiada osobne pole w klasie, lecz przekazywanie takich danych do funkcji jest bardzo niewygodne:

```
<?php
function rob($chodz, $mow, $rozmawiaj, $wstan, $idz_gdzies)
{
    if($chodz == 1){ echo "Kazales mi chodzic<br>"; }
    if($mow == 1){ echo "Kazales mi mowic<br>"; }
    if($rozmawiaj == 1){ echo "Kazales mi rozmawiac<br>"; }
    if($wstan == 1){ echo "Kazales mi wstac<br>"; }
    if($idz_gdzies == 1){ echo "Kazales mi isc gdzies<br>"; }
} // end rob();

rob(1,0,1,1,0);
?>
```

Czy wywołanie takiej funkcji wydaje Ci się przyjazne? Mamy szereg zer i jedynek bez żadnej informacji w kodzie, co każda z nich oznacza. Co więcej, gdy zajdzie potrzeba dodania nowego parametru, jesteśmy w kropce. Musimy odnaleźć wszystkie wywołania i każde z nich pozmienić, by uwzględniło istnienie nowego stanu.

W programowaniu w językach kompilowanych funkcje z dużą liczbą argumentów rodzą jeszcze jeden problem, mianowicie ich wywołanie trwa dłużej, jako że każdy argument musi być z osobna odłożony na stos. Zwróć jednak uwagę, że możemy to przyspieszyć, szczególnie w PHP. Do funkcji przekazujemy szereg liczb 0 i 1, które są zapisywane jako 32-bitowe liczby całkowite ze znakiem, pomimo iż tak naprawdę potrzebny jest nam tylko jeden bit do zapisania potrzebnej informacji. Pozostałe 31 jest dla nas przy takim podejściu straconych i niepotrzebnie zapycha pamięć.

Problem jest łatwy do rozwiązania, jeżeli zastosujemy operatory bitowe pozwalające nam operować na pojedynczych bitach liczby. Za ich pomocą implementuje się tak zwane *flagi*. Zauważmy, że każda informacja, również liczba, zapisywana jest w postaci zerojedykowej. W przypadku liczby każdy bit

odpowiada kolejnej potęgze dwójki i tak np. liczba 32 to szósty bit, a 256 - dziewiąty (pierwszy bit odpowiada zerowej potęgze dwójki, stąd lekko przesunięta numeracja). Możemy więc każdemu stanowi przyporządkować określone potęgi i za ich pomocą identyfikować odpowiednie bity naszej liczby. Zaczniemy więc od tego kroku:

```
define('WALK', 1);
define('TALK', 2);
define('SPEAK', 4);
define('WAKE_UP', 8);
define('GO_SOMEWHERE', 16);
```

Za pomocą stałych oznaczyliśmy pięć pierwszych bitów liczby. Teraz wkraczają do akcji nam operatory bitowe, które pozwolą nam sprawdzić, czy dany bit jest ustawiony oraz wygenerować pewną kombinację. Porzucmy funkcję i skorzystajmy ze zmiennej. Załóżmy, że chcemy ustawić stany *WALK*, *SPEAK* oraz *GO_SOMEWHERE*. Do tego posłużą nam operator sumy logicznej |

```
$zmienna = WALK | SPEAK | GO_SOMEWHERE;
```

Zwróćmy uwagę, że teraz dzięki stałym wyraźnie widać, co tak naprawdę ustawiamy. Co więcej, sumowanie logiczne jest przemienne, więc kolejność ustawiania flag może być dowolna. Wynikiem działania jest pewna liczba opisana przez ustawione bity. Aby sprawdzić, jakie stany zażyczył sobie użytkownik, przyda nam się operator koniunkcji bitowej &

```
if($zmienna & GO_SOMEWHERE)
{
    echo "Kazales mi gdzies isc<br>";
}

if($zmienna & WAKE_UP)
{
    echo "Kazales mi wstac<br>";
}

if($zmienna & SPEAK)
{
    echo "Kazales mi mowic<br>";
}

if($zmienna & TALK)
{
    echo "Kazales mi rozmawiac<br>";
}

if($zmienna & WALK)
{
    echo "Kazales mi chodzic<br>";
}
```

Operator ten w tym konkretnym wypadku zwraca nam 1, jeśli dany bit jest ustawiony na 1 w zmiennej *\$zmienna*, jednak jego zastosowanie jest szersze. Załóżmy, że mamy dwie konfiguracje flag: *\$konf1* oraz *\$konf2*. Aby poznać, jakie flagi są ustawione jednocześnie w obu konfiguracjach, także przyda nam się ten operator:

```
$wynik = $konf1 & $konf2;
```

Jak myślisz, co będzie wynikiem następującego działania:

```
$wynik = $konf1 | $konf2;
```

Flagi bitowe to bardzo pożyteczne narzędzie oszczędzające miejsce, eleganckie w użyciu i wydajne (porównania bitowe realizowane są sprzętowo przez procesor). Są także odporne na dorzucenie nowych stanów. Wystarczy po prostu zadeklarować nową stałą i dopisać kod obsługi, natomiast nie jest konieczna modyfikacja wszystkich już istniejących wywołań. PHP ma tylko jedno ograniczenie. Ponieważ wszystkie liczby są 32-bitowe, oczywistym jest, że ilość flag, jakie można zapisać w jednej liczbie, jest w naturalny sposób ograniczona do 32. Poczynam jednak, że sytuacje, gdy potrzeba ich więcej, są naprawdę rzadkie.

Nowe zastosowanie operatorów logicznych

Operatory logiczne zazwyczaj wykorzystywane są przy instrukcji warunkowej do oznaczenia, że np. dwie relacje muszą zajść równocześnie:

```
<?php
    if($a > 4 && $a < 8)
    {
        // ...
    }
?>
```

Jednak ich działanie jest bardzo podobne do poznanych już operatorów bitowych, z tym że traktują one podaną wartość jako całość i nie ingerują w pojedyncze bity. Można to wykorzystać w systemie uprawnień, gdzie użytkownik może być członkiem wielu grup. Grupy mogą mieć różne uprawnienia i niekiedy zachodzi sytuacja, że do zasobu X próbuje się dostać człowiek, który jednocześnie ma prawo do wejścia (gdyż należy do grupy A) oraz go nie ma (zgodnie z zasadami grupy B). Operatory logiczne mogą nam tu pomóc w narzuceniu odpowiedniej polityki. Jeśli pragniemy, aby dostęp był możliwy wyłącznie, gdy wszystkie grupy danego użytkownika zezwalają na to, stosujemy operator **&&** do wyprodukowania stanu wynikowego:

```
<?php
    $grupy = array(
        'grupa1' => array('zasob' => 0),
        'grupa2' => array('zasob' => 1),
        'grupa3' => array('zasob' => 0)
    );
    $wynik = array('zasob' => 1);
    foreach($grupy as $grupa)
    {
        $wynik['zasob'] = $wynik['zasob'] && $grupa['zasob'];
    }
?>
```

Początkową wartością stanu wynikowego musi być 1, gdyż inaczej użytkownik nie zostanie wpuszczony nawet, gdy wszystkie grupy mu na to zezwolą (spróbuj prześledzić, dlaczego, wykonując kolejne operacje logiczne). Dlatego też musisz uważać i w sytuacji, gdy użytkownik nie należy do żadnej grupy, ręcznie zresetować stan na 0.

Jeśli zamienimy operator **&&** na **||**, uzyskamy sytuację, że przynajmniej jedna grupa musi przepuszczać użytkownika, aby dostał on dostęp do zasobu. W tym wypadku początkową wartością wynikową musi być 0. Co więcej, możemy pokusić się, aby po wykryciu pierwszego stanu z jedynek przerwać pętlę, gdyż dalsze sprawdzanie jest już zbędne.

W ramach ćwiczenia spróbuj sprawdzić, co się stanie, gdy użyjemy trzeciego operatora logicznego – **xor**, interpretując jego wyniki.

Instrukcja warunkowa na operatorach

W PHP istnieją dwie wersje operatorów logicznych. Symboliczne, np. **&&** oraz tekstowe, np. **and**. Robią one dokładnie to samo, lecz nie wszyscy wiedzą, że między nimi jest pewna różnica. Otóż każdy z tych operatorów ma różne priorytety. Wersje symboliczne są wysoko, przez co sąsiednich wyrażeń nie trzeba otaczać nawiasami. Operatory tekstowe mają znacznie niższy priorytet:

```
<?php
    if($a > 4 && $a < 8){ /* ... */ }
    if( ($a > 4) and ($a < 8) ){ /* ... */ }
?>
```

Z powodu różnic w priorytetach, w drugim przypadku sąsiednie wyrażenia musiały być otoczone nawiasem. Za dawnych czasów właściwość ta była w interesujący sposób używana do obsługi błędów połączenia z bazą MySQL:

```
<?php
    mysql_connect('localhost', 'root', '') or die('Błąd!');
?>
```

Na dłuższą metę sposób był niewygodny, gdyż o wiele prościej było napisać sobie nakładkę z wbudowaną obsługą błędów, ale sposób wykorzystania operatora jest interesujący. Właściwość tę możemy wykorzystać do usunięcia z naszego kodu części instrukcji warunkowych. Przypuśćmy, że zależy nam na wykonaniu pewnego kodu, gdy pewna opcja jest włączona:

```
<?php
    if($config['opcja'])
    {
        akcja();
    }
?>
```

Alternatywny sposób zapisania wygląda następująco:

```
<?php
    $config['akcja'] and akcja();
?>
```

Dlaczego to działa? Otóż interpreter PHP jest na tyle sprytny, że nie liczy wyrażen do końca, jeśli już wcześniej można bezbłędnie przewidzieć, jaki będzie wynik. Zauważmy, że jeśli `$config['akcja']` będzie równa 0, bez względu na wynik funkcji `akcja()`, możemy już powiedzieć, że całe wyrażenie da nam 0. Z tego powodu funkcja nie zostanie wtedy wykonana. Gdy zaś opcja wynosi 1, PHP wykonuje funkcję, ponieważ jest to potrzebne do określenia całego wyrażenia. Operator więc działa nam jak swego rodzaju warunek.

Oczywiście taki zapis ma pewne ograniczenia – po prawej stronie nie możemy mieć klamerek, ani żadnej struktury kontrolnej. Poniższy kod spowoduje błąd przy próbie uruchomienia (a szkoda):

```
<?php
    $config['akcja'] and throw new Exception;
?>
```

Zapis tego rodzaju często wykorzystuję w moim systemie szablonów Open Power Template. W pewnym miejscu mam dość zagmatwany kawałek kodu, który musi dodatkowo wszystkim przetwarzanym w pętli obiektom przekazać pewną wartość, jeśli ustawiony jest pewien wyliczony na początku status. Dorzucenie kilku **ifów** jeszcze bardziej by wszystko skomplikowało, a tak mam tylko:

```
$stateSet and $item->set('something', $value);
```

Zakończenie

PHP, podobnie jak wiele innych języków, wiele istotnych rzeczy oblicza właśnie dzięki wyrażeniom, dlatego czas zainwestowany w dobre poznanie i zrozumienie filozofii ich działania na pewno zaprocentuje w przyszłości. Umiejętnie nimi się posługując, można krótko i zwięźle zapisać obliczenia, które normalnie wymagałyby sporo linijek kodu. Co więcej, wiedza ta jest uniwersalna i przyda się nie tylko w PHP, ale i w

JavaScriptcie, C++, Javie i wielu innych językach.

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na www.zyxist.com

Licencja

Artykuł rozpowszechniany jest na licencji **Creative Commons Uznanie autorstwa-Użycie niekomercyjne-Bez utworów zależnych 2.5 Polska**.

Wolno:

- kopiować, rozpowszechniać, odtwarzać i wykonywać utwór

Na następujących warunkach:

1. *Uznanie autorstwa*. Utwór należy oznaczyć w sposób określony przez Twórcę lub Licencjodawcę*.
2. *Użycie niekomercyjne*. Nie wolno używać tego utworu do celów komercyjnych.
3. *Bez utworów zależnych*. Nie wolno zmieniać, przekształcać ani tworzyć nowych dzieł na podstawie tego utworu.

W celu ponownego użycia utworu lub rozpowszechniania utworu należy wyjaśnić innym warunki licencji, na której udostępnia się utwór.

Każdy z tych warunków może zostać uchylony, jeśli uzyska się zezwolenie właściciela praw autorskich.

Powyższe postanowienia w żaden sposób nie naruszają uprawnień wynikających z dozwolonego użytku ani żadnych innych praw.

Internetowa skrócona wersja licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/>

Pełen tekst licencji:

<http://creativecommons.org/licenses/by-nc-nd/2.5/pl/legalcode>

Podczas publikacji należy podać następujące informacje:

1. Link do internetowej skróconej wersji licencji.
2. Informację o wersji publikowanego tekstu.
3. Informacje o autorze w następujący sposób:

Tomasz "Zyx" Jędrzejewski

Aktualna wersja artykułu zawsze na www.zyxist.com