instadb (print.php?report_id=wiki_page&id=454) -> Nianio Lang (print.php?report_id=wiki_page&id=478) -> NianioLang

# NianioLang

NianioLang
PLE 2012-06-20

Kolorowanie składni w NanioLang dla vim
Instrukcja na VIM (print.php?report_id=wiki_page&id=560)

Co jest w perlu, a czego nie ma i nie będzie w nianio_lang:

- referencje, obiekty, argumenty opcjonalne dla funkcji, przyjmowanie zbioru argumentów jako tablicy przez funkcję(znak @ w prototypie funkcji perlowej).

**Niezmienniczość struktur w nianio_lang:**
Struktury w nianio lang są niezmiennicze, jeżeli przekażemy jakiś argument do funkcji, to ta funkcja nie ma prawa zmienić wartości naszego argumentu.
Przy czym możemy bez problemu używać standardowej składni imperatywnej. Wszelkie zmiany załatwi wtedy nianio lang:
np.

```
def funkcja(a) {
    a[0] = 22;
    nl::print(a[0]);
}

def funkcja2() {
    var b = [33];
    funkcja(b);
    nl::print(b[0]);
}
```

Wywołanie funkcja2() spowoduje wypisanie kolejno 22 i 33.

**Jakie struktury akceptuje nianio_lang:**

- łańcuchy znakowe -> 'ala ma kota'
- liczby -> 22
- tablice -> [22, 'ala ma kota']
- hashe -> {klucz => 44, klucz2 => 'ala ma kota'}
- kombinacje powyższych

**L-wartość:**
L-wartością w NL jest:

- wyrażenie po lewej stronie przypisania
- wyrażenie po lewej stronie operatora pryzpisaniopodobnego(+=, -=, /=, .=, etc.)
- argument ref'a
  L-wartość w NL jest postaci:
- zmienna
- odwołanie do indeksu czegoś, co jest L-wartością(np. a[0], b->c[0])
- odwołanie do wartości w hashu czegoś co jest L-wartością (np. a{zmienna}, a[0]{zmienna})
- odwołanie do klucza w rekordzie, który jest L-wartością(np. b->c, b[0]->d, b->c->d->e)

**Aktualna składnia(może się zmienić):**

```
use nazwa_modulu; -> use jak w perlu
var i; - deklaracja zmiennej
var h = {klucz => 'wartosc', k2=> 'wartosc2'}; - inicjali
zacja hasha
var arr = [1, 2, 3, 4]; - inicjalizacja tablicy
nazwa_hasha->nazwa_pola = 2; - odwołanie do pola w hashu
arr[2]; - odwołanie do elementu w tablicy
h{zmienna} - odwołanie do wartości w hashu poprzez klucz.
break;
continue; - wiadomo

def nazwa_modulu::nazwa_funkcji() { - definicja funkcji p
ublicznej
}
def nazwa_funkcji() { - definicja funkcji prywatnej
}

def nazwa_funkcji(ref element) { - definicja elementu będ
ącego refem  - patrz imref w grids
}

nazwa_modulu::nazwa_funkcji(argument1, argument2) - wywol
anie funkcji,
```

wywołania funkcji mieć formę nazwa_modulu::nazwa_funkcji z obowiązkową nazwą modułu.
Podanie nazwy funkcji bez nazwy modułu zawsze oznaczana funkcję prywatną dla modułu.

instrukcje sterujące:

```
if  (warunek)  {
} elsif (warunek2) {
} else {
}
```

jednolinijkowe:

```
--i if warunek;
++i unless warunek;
++i while i < 10;
```

Jak również rep. fora, forh, for etc.

instrukcja znajdująca się przed taką pętlą musi być jednym z poniższych:

- wyrażenie
- die
- return
- try/ensure
- deklaracja zmiennej

**Operatory:**
W NianioLangu są dostępne następujące operatory:

- unarne
  - prawostronnie łączne
    - @
    - !
    - +
    - -
  - bez możliwości kaskadowego użycia
    - ++

---

- binarne
  - lewostronnie łączne
    - ->
    - is
    - as
    - *
    - /
    - %
    - +
    - -
    - .

        - &&
        - ||
        - []
        - {} - jako odwołanie do hasha
    - prawostronnie łączne
        - =
        - +=
        - /=
        - *=
        - .=
        - []= - operator push na tabllicy
    - bez możliwości kaskadowego użycia
        - <
        - <=
        - >
        - aaaa >=
        - ==
        - !=
        - eq
        - ne
- ternarne
    - prawostronnie łączne
      *?:
      Operatory <, <=, >, >=, ==, != służą do porównywania liczb. Natomiast operatory eq i ne umożliwiają porównywanie napisów.

**ref:**
Ref działa tak: po wywołaniu funkcji argumentowi zostanie przypisana ostatnia wartość zmiennej lokalnej w funkcji, do której został przypisany argument.
Argumentem ref'a jest l-wartość
Na przykładzie:

```
def funkcja(ref a) {
    a[0] = 22;
}

def funkcja2() {
    var a = [33];
    funkcja(ref a);
    nl::print(a[0]);
    var b = {c => [0, 0]};
    funkcja(ref b->c};
    nl::print(b->c[0]);
}
```

Wywołanie funkcja2() spowoduje wypisanie 22 i 22.

**Wielokrotny ref:**
W przypadku przypisania tej samej zmiennej do kilku refów, do zmiennej zostanie przypisany ostatni ref(w kolejności argumentów funkcji).
Np.

```
def funkcja(ref a, ref b) {
    a = 10;
    b = 20;
    ++a;
}

def funkcja2() {
    var aa = -1;
    funkcja(ref aa, ref aa);
    nl::print(aa);
}
```

Wywołanie funkcja2() spowoduje wypisanie: 20 na ekran.

**Referencje do funkcji**
Funkcja w NL może przyjmować referencję do funkcji jako parametr.
Np.

```
def funkcja(arg1) {
}
```

Wtedy wywołanie wygląda np. tak:
funkcja(@nazwa_modulu::nazwa_funkcji);

**warianty:**7

```
var a = :varaint('cos'); - wartiant z argumentem
var b = :nazwa_wariantu = wariant bezargumentowy


match (variant) case :var1(var el) {
} case :variant2(var el2) {
} case :variant3 {
} - jeżeli match nie natrafi na żaden z wariantów to pole
ci die;
```

match służy do rozpakowywania wariantów, case'y powinny pokrywać wszystkie możliwe wartości wariantu.
Przykład:

```
match(:var1('cos')) case :var1(var el) {
    nl::print(el);
} case :var2 {
}
```

wypisze: cos

**try i ensure: - MZU 2012-10-15**

nowym tworem w nianiolangu są komendy try i ensure.
służą one do obsługiwania błędów - sytuacji nietypowych.

tak samo używamy ensure jak try:

1. try var a = f(); # można przypisać dowolne wyrażenie zwracające odpowiedni typ
2. try a = f(); -- | | --
3. try f(); -- | | --

Funkcja f() musi zwrócić wartość zgodną z typem:
ptd::var({ok => ptd::ptd_im(), err => ptd::ptd_im()})
jeśli wszytko poszło dobrze i chcemy dalej wykonywać kod należy zwrócić wariant "ok", np.:
:ok(struktura)
w wyniku tego do zmiennej a zostanie przypisane wartość struktury.
dla przypadku 3 po prostu kod będzie się dalej wykonywać.

W przypadku błędy zwracamy wariant "err". Działanie w zależności od komendy:
try - ponieważ w nianio kod zawsze się wykonuje w funkcji zostanie przerwane wykonanie i zostanie zwrócone to co otrzymaliśmy czyli :err(ptd::ptd_im()).
ensure - zabija gridsy a jako argument die wstawia zawartość err.

Poniżej jest kod robiący to samo co: try a = f();

```
    match(f())case :ok(var ok){
        a = ok;
    } case :err(var err){
        return :err(err);
    }
```

dla ensure wygląda podobnie:

```
    match(f())case :ok(var ok){
        a = ok;
    } case :err(var err){
        die(err);
    }
```

**inne:**

die; - standardowo jak perlu

funkcje biblioteczne do operowania na hashach tablicach itp są w katalogu nianio_lib

pętle:

```
fora var elem (tablica) {
{
forh var key, var value (hash) {
}
while(warunek) {
}
loop { - petla nieskonczona
}
rep var i (10) {
}  - to samo co for(var i = 0; i < 10; ++i) -
- tej pętli należy używać w większości przypadków, któryc
h w innym języku użylibyśmy for'a

for (var i = 10; i > 0; i -= 3) {
}
```

Gwarancje kolejności iteracji dla poszczególnych rodzajów pętli:

- fora - kolejność iteracji jest gwarantowana: jest zawsze taka sama jak elementów w tablicy
- forh - kolejność iteracji jest dowolna, brak jakichkolwiek gwarancji (2016-10-19 PLE TODO leksykograficznie)
- rep - kolejność iteracji jest gwarantowana: dla rep var i (n) zmienna i w kolejnych iteracjach przyjmuje zawsze wartości w kolejności od 0 do n - 1

**Referencje do funkcji:**
Referencję do funkcji można pobrać przez @nazwa_modulu::nazwa_funkcji. Referencję można pobrać tylko do funkcji publicznych dla modułu.

**Łańcuchy znakowe:**

- Poprawny format napisu: 'cos ciekawego'. Escapowanie przez apostrof np. 'ala '' ma kota' tłumaczy się na "ala ma ' kota". Znaki specjalnie dostępne w module string z biblioteczki.

**Boolean:**

- W nianio lang odpowiednie instrukcje oczekują jako argumentów typów logicznych tj. true lub false, które są aliasami na warianty :TRUE lub :FALSE.
  Przykład:
  ```
  var a = true;
  if (false || (true && 2 == 2)) {
  }
  ```
- boolean::TRUE, boolean::FALSE, to odpowiedniki w perlu.

W nianio langu if, unless, itp. oraz operatory logiczne np. || przyjmują booleany jako argument a nie np. liczby całkowite.

**Zmienne/stałe globalne dla modułu:**

- W kolejnej wersji NL nie będzie zmiennych globalnych dla modułu. Dostępne będą jedynie stałe o widoczności dla całego modułu.
- Próba modyfikacji stałych powinna zakończyć się błędem kompilacji.

**Komentarze:**

- Komentarzem jest wszystko po znaku # do końca linii. Komentarze wylatują przy pretty printingu.

**Formatowanie kodu w nl:**
Zasady kodowania w NianioLangu (print.php?report_id=wiki_page&id=500)