Ciekawe pętle i iteracje na drugą nóżkę

Paweł Jackowski
GUST
P.Jackowski@gust.org.pl

Streszczenie

Do czego to podobne, żeby programista musiał sam sobie zaimplementować pętle? Do TEX-a! TEX jako język programowania podobny jest tylko do siebie. Ciekawą własnością TEX-a, rzadko spotykaną wśród języków programistycznych, jest brak wbudowanej pętli. Jednak dzięki temu, iż TEX doskonale znosi definicje rekursywne i potrafi sprawdzać warunki, nie ma przeszkód, by pętle definiować samodzielnie. Zrobił to Donald Knuth w `plain`-ie, poprawiał Alois Kabelschacht, Kees van der Laan, Marcin Woliński i wielu innych, a używa każdy praktykujący TEX-owiec. Artykuł podsumowuje to, co każdy TEX-owiec o pętlach wiedzieć powinien, nie stroniąc od kruczków i sztuczków, o których wiedzieć nie musi.

Czepianie się `\loop`. Przyjrzyjmy się raz jeszcze tradycyjnej, `plain`-owej definicji pętli ([1], str. 352):

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body \let\next\iterate
 \else \let\next\relax\fi \next}
\let\repeat=\fi
```

Definicja jest dość czytelna i zrozumiała dzięki makrom pomocniczym o sugestywnych nazwach `\next` oraz `\body`. Wykonujemy tutaj jednak niepotrzebne przypisanie przy każdej iteracji. Przypisanie to nadaje znaczenie instrukcji `\next`, której, tak samo jak instrukcji `\body`, w zasadzie nie powinniśmy używać gdziekolwiek indziej. Ponieważ jednak instrukcje te są schowane przed użytkownikiem, łatwo o konflikt nazw.

Powszechnie znanych jest kilka ulepszeń tradycyjnej konstrukcji, wykorzystujących polecenie `\expandafter` zamiast makra pomocniczego `\next`. Na przykład ([2]):

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body\expandafter\iterate\fi}
```

albo jeszcze prościej ([6]);

```
\def\loop#1\repeat{%
 \def\iterate{%
  #1\expandafter\iterate\fi}%
 \iterate}
```

W pierwszym przypadku pozbywamy się zbędnej definicji `\next`, w drugim także definicji `\body`. W jeszcze innej konstrukcji (opisanej szerzej w [3]) cała zawartość pętli wykonywana jest poza blokiem warunku `\if...\fi`:

```
\def\loop#1\repeat{%
 \def\body{#1}\iterate}
\def\iterate{%
 \body\else\etareti\fi\iterate}
\def\etareti\fi\iterate{\fi}
```

Podsumowanie tych i innych rozwiązań można znaleźć w [4].

Pętla w pętli. Powyższe konstrukcje, choć poprawne i eleganckie, nie umożliwiają stosowania pętli zagnieżdżonych. We wszystkich pierwszą operacją jest zapamiętanie zawartości pętli w jakiejś instrukcji. Zagnieżdżenie spowodowałoby konflikt znaczeń tej instrukcji między pętlą zewnętrzną i wewnętrzną.

Jak z tego wybrnąć? Kosztem nieznacznego spowolnienia pętli, można zastosować parametr makra w miejsce definicji. Przykładowo, zamiast powtarzać instrukcję `\body` w każdym `\iterate`, możemy zadać fragment powtarzanego kodu jako parametr instrukcji `\iterate`. Dla wygody użytkowej stosujemy prefiks `\long`, aby w pętli mogła znaleźć się instrukcja `\par`. Definiujemy także makro `\gobbleone`, które wykonywane jest przed samym wyjściem z pętli i pochłania nadmiarowy parametr znajdujący się zaraz za `\fi` kończącym warunek.

```
\long\def\loop#1\repeat{%
 \iterate\gobbleone{#1}}
\long\def\iterate\gobbleone#1{%
 #1\expandafter\iterate\fi
 \gobbleone{#1}}
\long\def\gobbleone#1{}
```

Definicja `\gobbleone` pełni tutaj dodatkową rolę – stanowi ogranicznik makra `\iterate` (ang. *macro* *delimiter*). Kiedy wykonywana jest instrukcja `\iterate`, następujące po niej `\gobbleone` jest pochłaniane jako (nieużywany) fragment parametru. Na końcu pętli instrukcja `\iterate` jest pomijana, za to `\gobbleone` pochłania parametr z zawartością pętli.

Powyższa pętla może być używana tak samo jak jej postać tradycyjna, z tą różnicą, że pętle można zagnieżdżać. Pokazuje to przykład:

```
\count100=9
\loop{\count101=65 % ASCII ‘A’
 \advance\count100 by-1
 \ifnum\count100>0
 \leavevmode\loop
  \char\count101 \the\count100
  \advance\count101 by1
  \ifnum\count101<73 \space
\repeat\par
}\repeat
```

Zastosowanie tego kodu da coś w rodzaju pól szachowych. Elementy każdego wiersza składa pętla wewnętrzna, zaś całe wiersze pętla zewnętrzna.

```
A8 B8 C8 D8 E8 F8 G8 H8
A7 B7 C7 D7 E7 F7 G7 H7
A6 B6 C6 D6 E6 F6 G6 H6
A5 B5 C5 D5 E5 F5 G5 H5
A4 B4 C4 D4 E4 F4 G4 H4
A3 B3 C3 D3 E3 F3 G3 H3
A2 B2 C2 D2 E2 F2 G2 H2
A1 B1 C1 D1 E1 F1 G1 H1
```

Należy zwrócić uwagę na zastosowanie grupy w zewnętrznym bloku pętli:

```
\loop{...\loop...\repeat...}\repeat
```

Grupa ta ma tutaj wpływ tylko na zasięg czytanego parametru. Sama zawartość pętli nadrzędnej nie jest wykonywana w grupie. Dzięki temu pętla zewnętrzna może bez przeszkód korzystać z przypisań wykonanych w pętli wewnętrznej. Zastosowanie grupy jest konieczne – bez niej TEX skończyłby czytać zawartość nadrzędnej pętli natychmiast po pierwszym napotkanym `\repeat`.

Niechże się rozwija. Możliwości TEX-a nie kończą się na inkrementacji i sprawdzaniu wartości licznika. Również TEX-owe iteracje nie kończą się na konstrukcjach `\loop...\repeat`. Często istnieje na przykład potrzeba wykonania jakiejś procedury kolejno dla wszystkich leksemów pewnej grupy i to w takim kontekście, w którym nie możemy używać przypisań (podczas tworzenia definicji z użyciem `\edef`, `\xdef`, wewnątrz `\write`-ów, `\special`-i oraz `\mark`-ów). Tutaj warto przytoczyć piękne w swej prostocie makro `\fifo`, opisane szerzej w [3]:

```
\def\fifo#1{\ifx\ofif#1\ofif\fi
 \process#1\fifo}
\def\ofif#1\fifo{\fi}
```

A oto przykład użycia `\fifo`, w którym „w locie” tworzymy ściągę z kodów polskich znaków:

```
\def\process#1{(#1 -> \number‘#1)}
\immediate\message
 {\fifo ąćęłńóśżź\ofif}
```

Tutaj zamiast zliczać powtórzenia, wykonujemy dla kolejnych parametrów instrukcję `\process`. Na początku każdej iteracji warunek `\ifx` sprawdza, czy właśnie połknięty parametr nie jest leksemem `\ofif`. Ten ostatni występuje tutaj jednocześnie w roli ogranicznika listy leksemów oraz makra kończącego warunek, wykonywanego po ostatniej iteracji.

Gry liczbowe. Nikogo nie trzeba przekonywać, że makr rozwijalnych (bez przypisań) używa się wygodniej. A jak ominąć przypisania w pętlach operujących na liczbach? W końcu najbardziej typowe użycie pętli to powtórzenie jakiegoś fragmentu kodu określoną liczbę razy. Pokazane wcześniej konstrukcje `\loop...\repeat` osiągają to poprzez iteracyjne zwiększanie (lub zmniejszenie) pewnego licznika, a to wymaga przypisań.

Sprawa nie jest jednak beznadziejna. Jak pokazuje ostatni przykład, instrukcja `\number` rozwija „w locie” dowolną TEX-ową reprezentację liczby do postaci dziesiętnej. W podstawowej wersji TEX-a każda operacja arytmetyczna wymaga już przypisania. Z pomocą może przyjść ε-TEX udostępniający kilka wygodnych operacji pozwalających unikać niewygodnych przypisań. Za przykład posłuży tu instrukcja `\numexpr` dokonująca podstawowych operacji na liczbach (dodawanie, odejmowanie, mnożenie i dzielenie) w sposób rozwijalny.

Spróbujmy za pomocą omawianej instrukcji zbudować makro `\replicate`, które powtarza dowolny fragment kodu określoną liczbę razy. Pierwszy parametr makra to liczba powtórzeń, drugi zaś to zawartość pętli.

```
\long\def\replicate#1#2{%
 \ifnum\numexpr#1>0
  #2\replicate{#1-1}{#2}\fi}
```

Pętlę zaczynamy od sprawdzenia, czy licznik jest dodatni, tj. czy należy wykonać powtórzenie. Jeśli tak, wykonywana jest zawartość pętli podana jako drugi parametr, po czym następuje rekursywne wywołanie procedury `\replicate` z licznikiem zmniejszonym o 1 i z drugim parametrem w niezmienionej postaci.

Taka konstrukcja ma dwie poważne wady. Po pierwsze, każde kolejne powtórzenie następuje wewnątrz kumulujących się bloków `\ifnum...\fi`, co grozi katastrofą przy dużej liczbie powtórzeń. Po drugie, pierwszy parametr makra w każdym obrocie pętli powiększa się o dwa znaki, a podczas sprawdzania wartości licznika, TEX zmuszony jest każdorazowo do wyliczenia coraz dłuższego wyrażenia postaci `\numexpr100-1-1-1....`

Spróbujmy zatem tak zmodyfikować makro `\replicate`, aby każde powtórzenie wykonywane było poza warunkiem `\ifnum...\fi`, oraz żeby parametr reprezentujący licznik miał bardziej elegancką postać.

```
\long\def\replicate#1#2{%
 \ifnum\numexpr#1>0
  #2\expandafter\replicate\expandafter
  {\number\numexpr#1-1\expandafter}%
 \else
  \expandafter\gobbleone
 \fi{#2}}
```

Znów zaczynamy od sprawdzenia, czy licznik pętli jest dodatni, tj. czy należy wykonać powtórzenie. Jeśli tak, wykonywana jest zawartość pętli (drugi parametr), po której `\expandafter` w połączeniu `\number\numexpr` zmniejsza licznik o jeden i ponownie uruchamia procedurę `\replicate` z nową wartością licznika. Drugi parametr procedury `\replicate` przekazywany jest bez zmian i znajduje się za instrukcją `\fi` kończącą warunek. Kiedy licznik osiągnie 0 (lub kiedy złośliwie uruchomimy pętlę z parametrem nie większym niż zero) `\expandafter` unicestwia pozostałe `\fi`, po czym uruchamiana jest znana już procedura `\gobbleone` połykająca nadmiarowy parametr.

Korzystamy tu ze wspomnianej, zbawiennej własności instrukcji `\number`, która powoduje rozwijanie makr następujących po niej do końca, tj. do dziesiętnej reprezentacji liczby. Podczas rozwijania wyrażenia `\numexpr`, wykonywana jest instrukcja `\expandafter`, która, niby mimochodem (podczas rozwijania liczby!), powoduje zniknięcie bloku warunku. Następnie TEX „orientuje się”, że wyrażenia nie da się dalej rozwinąć i powraca do instrukcji `\replicate`. Ta ostatnia uruchamiana jest z liczbowym parametrem w dziesiętnym zapisie i z drugim parametrem stanowiącym niezmienną zawartość pętli. Dzieje się to już poza blokiem warunku.

A oto przykład użycia makra `\replicate` w kontekście, w którym zawiodłaby tradycyjna pętla TEX-owa wykorzystująca przypisania. Makro `\replicate` jest rozwijalne i można je zagnieżdżać.

```
\immediate\message
 {\replicate{100+1}
  {Będę używał eTeX-a%
    \replicate{3}{!} }}
```

Postawmy sobie teraz mniej szkolne zadanie. Spróbujmy zdefiniować makro `\fixed`, które wstawia cyfrę 0 przed inne cyfry tak, aby uzupełnić zapis liczby do określonej długości. Przykładowo,

```
\fixed{4}{12}
```

ma się rozwinąć do `0012`. Zacznijmy od makra pomocniczego służącego do „mierzenia” długości napisów.

```
\long\def\abacus#1{\addabacus#10}
\long\def\addabacus#1#2#3{%
 \ifx#3#1#2\else
  \expandafter\addabacus
  \expandafter#1\expandafter
  {\number\numexpr#2+1\expandafter}%
 \fi}
```

Makro `\abacus` (z łac. *liczydło*) zlicza leksemy występujące między parą dowolnych innych leksemów.

```
\count100=\abacus|Konstantyno%
 politańczykiewiczówna|
\edef\numofletters{%
 \abacus\relax Antiestablish%
  mentarianism\relax}
```

Przy każdym obrocie pętli makro sprawdza, czy kolejnym leksemem połykanym jako parametr jest ten zastosowany jako ogranicznik mierzonego napisu. Jeśli nie, makro w znany już sposób podnosi o 1 swój licznik i wykonuje kolejną iterację. Jeśli tak, makro po prostu zwraca swój licznik, będący liczbą leksemów znajdujących się między dowolnie wybranymi ogranicznikami.

Teraz korzystając z makr `\replicate` oraz `\abacus` definiujemy makro uzupełniające napis wybranym znakiem, tak aby uzyskać napis określonej długości.

```
\def\fixedprefix#1#2#3{%
 \expandafter\replicate\expandafter
 {\number
   \numexpr#1-\abacus\relax#2\relax}
 {#3}#2}
```

Jeśli teraz napiszemy

```
\edef\test{\fixedprefix{4}{ab}{*}}
```

w instrukcji `\test` zostanie zapamiętany napis `**ab`. Pozostaje skonstruować wyspecjalizowaną postać makra `\fixedprefix`, która formatuje liczby tak, aby składały się z określonej liczby cyfr, uzupełniając w razie potrzeby zerami. Ponieważ makro `\fixed` ma operować na liczbach, pierwszą operacją, jaką powinniśmy wykonać, jest rozwinięcie parametrów podanych przez użytkownika do ciągu samych cyfr. Tę sztuczkę też już znamy.

```
\def\fixed#1#2{%
 \expandafter\fixedzero\expandafter
  {\number\numexpr#1\expandafter}%
 \expandafter{\number\numexpr#2}}
\def\fixedzero#1#2{%
 \fixedprefix{#1}{#2}{0}}
```

Wiemy już, że TEX wytrwale i do końca rozwija liczby. Wiemy także, że bez trudu radzi sobie z długimi ciągami leksemów pożeranych jako parametry. Przedstawione poniżej makro `\rnum` (skrót od ang. *read number*) wykorzystuje obie wspomniane TEXniki iteracji do czytania liczb w różnej notacji, od dwójkowej do szesnastkowej.

```
\def\rnum#1#2{\dornum{#1}{0}#2\relax}
\def\dornum#1#2#3{\ifx#3\relax#2\else
  \expandafter\dornum\expandafter
  {\number
    \numexpr#1\expandafter}\expandafter
  {\number
    \numexpr#1*#2+"#3\expandafter}%
 \fi}
```

W ten sposób nauczyliśmy TEX-a, czym jest na przykład 1 000 000 000 000 w zapisie dwójkowym:

```
\count100=\rnum{2}{1000000000000}
```

Czytelnik pewnie zwrócił uwagę na znak '`"`' użyty w przedostatniej linii makra `\dornum`. Jak wiadomo, dla TEX-a oznacza to „czytaj cyfry szesnastkowo”. Bez tego zabiegu TEX nie zrozumiałby cyfr notacji wyższych niż dziesiętna, tj. od `A` do `F`.

Na deser propozycja makra `\xnum` działającego odwrotnie do `\rnum`. Makro zamienia notację liczby z systemu dziesiętnego do innych systemów, od dwójkowego do szesnastkowego, oczywiście w sposób rozwijalny. Jeśli Czytelnik dobrnął do tego miejsca, nie powinien mieć trudności ze zrozumieniem poniższego kodu. Tym, co nie używają ε-TEX-a należą się jednak dwa wyjaśnienia.

1. Jeśli podczas obliczania `\numexpr` ε-TEX napotka leksem `\relax`, natychmiast kończy czytać wyrażenie, a samo `\relax` znika bez śladu.
2. Jeśli w wyrażeniu `\numexpr` występuje dzielenie niecałkowite, wynik zostaje zaokrąglony, nie zaś, jak w tradycyjnym TEX-u, sprowadzony do części całkowitej.

Więcej na temat konstrukcji ε-TEX-owych w [5].

```
\def\hexdigit#1{%
 \expandafter\hexdigits
 \number\numexpr#1\relax\relax}
\def\hexdigits#1\relax
{\ifcase#1
  0\or1\or2\or3\or4\or5\or
  6\or7\or8\or9\or A\or
  B\or C\or D\or E\or F\fi}
\def\xnum#1#2{%
 \expandafter\doxnum\expandafter
 {\number
  \numexpr#1\expandafter}\expandafter
 {\number\numexpr#2}}
\def\doxnum#1#2{%
 \ifcase
  \ifnum#2<\numexpr#2/#1*#1\relax
   0 \else1 \fi
 \expandafter\doxnumdown\or
 \expandafter\doxnumup\fi
 {#1}{#2}}
\def\doxnumdown#1#2{%
 \ifnum#1>#2 \else
  \expandafter\doxnum\expandafter
  {\number#1\expandafter}\expandafter
  {\number\numexpr#2/#1-1\expandafter}\fi
 \hexdigit{#2-(#2/#1-1)*#1}}
\def\doxnumup#1#2{%
 \ifnum#1>#2 \else
  \expandafter\doxnum\expandafter
  {\number#1\expandafter}\expandafter
  {\number\numexpr#2/#1\expandafter}\fi
 \hexdigit{#2-#2/#1*#1}}
% test
\count100=\rnum{2}{1000000000}
\immediate\message
 {\xnum{16}{\count100}}
```

Bibliografia.

[1] Donald E. Knuth: *The TEXbook*.

[2] Alois Kabelschacht: `\expandafter` vs. `\let` and `\def` in conditionals and a generalization of plain's `\loop`. TUGboat, Volume 8 (1987), No. 2, 184–185.

[3] Kees van der Laan: FIFO and LIFO sing the BLUes. Biuletyn GUST nr 4 (1992), 20–26.

[4] Marcin Woliński: O pewnych konstrukcjach warunkowych i iteracyjnych. Biuletyn GUST nr 7 (1996), 5–9.

[5] Peter Breitenlohner: The ε-TEX Manual, Version 2, February 1998, 9.

[6] Victor Eijkhout: The bag of tricks. TUGboat, Volume 21 (2000), No. 1, 91.