Lobster, łącze TEX-Perl

Krzysio Leszczyński

Abstract

A small *in statu nascendi* system named LOBSTER is presented. It is both a bunch of TEX macros and a small Perl program interfacing the TEX code to various external programs, it can run a program or Perl routine *without* finishing the TEX job.

This text is freely distributable according to *GNU General Public Licence version 2*. You can get both Polish and English version of this article from ftp:/ftp.camk.edu.pl/private/chris/Lobster

Wątek hydrobiologiczny

W Sobieszewie Mistrz powiedział, a w Pappendal powtórzył: TEX nie jest lwem, TEX jest ośmiornicą. Głowonogi te żywią się mniejszymi lub większymi stworzonkami morskimi spacerującymi po dnie. Kontynując ten wątek zacząłem pisać mały programik LOBSTER, czyli po polsku homar. Zadaniem skorupiaka tego sympatycznego jest stworzenie elastycznego łącza między dwoma programami: TEX-em i Perlem. Ten ostatni jest ogromnym systemem, który w zadziwiająco krótkim czasie opanował środowisko unixowe, prawie całkowicie wypierając takie języki jak awk czy sed. Perl posiada bardzo silne narzędzia współczesnego programowania takie jak: moduły, pakiety, rekurencyjne struktury danych, programowanie jednocześnie proceduralno-obiektowe oraz makrodefinicyjne. Te cechy w połączeniu z prostotą Perla i dużymi możliwościami obróbki tekstów czynią go idealnym partnerem dla TEX-a.

Prawdziwy Głowonóg, jakim TEX jest niewątpliwie lubi pracować z wieloma programami, tu jednak zdarzaja się kłopoty:

- Kiedy programów zewnętrznych jest więcej niż jeden, dane dla nich należy starannie rozdzielać.
- Niektóre dane mogą znajdować się wyłącznie w pliku transkrypcyjnym. Przykładem niech będą tu wszelkie operacje typu `\show...`, Niestety nie ma w TEX-u polecenia `\showto` *plik*.
- Nie istnieje możliwość wykonania programu zewnętrznego przed zakończeniem zadania TEX-owego.

Te problemy mogą być rozwiązane, jeśli między danymi produkowanymi przez TEX-a a programem usługowym umieścić dodatkową warstwę.

Uruchamianie Lobster-a

LOBSTER jest programem napisanym w języku Perl. Wywołanie LOBSTER-a wygląda następująco:

`$ lobster` [*opcje*] `jobname`[`.tex`]

Po takim wywołaniu LOBSTER dzieli się na dwa procesy, proces macierzysty zamienia się na proces o postaci:

tex &format jobname

zaś proces potomny obserwuje cały czas plik transkrypcyjny `jobname.log` wykonując zaszyte w nim polecenia. `&format` oznacza `plain`, lub inny określony opcją LOBSTER-a `--fmt=nazwa`.

Istotna jest tutaj jednoczesność pracy TEX-a i LOBSTER-a, TEX może bowiem zlecić zadanie i nie kończąc pracy poczekać na wynik, ma to istotne znaczenie w zadaniach wymagających bardzo wielu przebiegów np. w oblewaniu nieprostokątnych rysunków tekstem. TEX-em staje się proces macierzysty co zapewnia użytkownikowi pełną kontrolę nad TEX-em, nawet w chwili wystąpienia błędu, bez odwoływania się do nieludzkiego w swej funkcji polecenia systemowego `kill`.

Poprzedni akapit dotyczy oczywiście wyłącznie środowisk, gdzie jest możliwa wieloprocesowość, co praktycznie ogranicza nas do unixów. DOS jest na razie praktycznie wyłączony z tych zastosowań. Implementacji na takie systemy jak MS-Windows, NT, VMS nie ma, bo ich nie posiadam. Tak naprawdę to wszystko zależy czy w konkretnej implementacji języka Perl jest obecna funkcja *fork*( ).

W środowisku unixowym LOBSTER czyta plik transkrypcyjny wykorzystując specjalny rodzaj pliku – kolejkę (ang.*named pipe*). Własnością takiego pliku jest możliwość czytania tylko z początku pliku, a pisania tylko na końcu. Określenie *plik* jest w tym miejscu nadużyciem, gdyż kolejka nie jest związana z żadnym miejscem na dysku, dane zapisywane do kolejki, w naszym przypadku do `\jobname.log` przesyłane są bezpośrednio do LOBSTER-a. Jest to o tyle ważne, ze pliki transkrypcyjne zapisane w środowisku LOBSTER bywają olbrzymie i czasami może nam zabraknąć miejsca na dysku. LOBSTER przepisuje cały plik transkrypcyjny do pliku `\jobname.lbslog` wycinając z niego wszystkie LOBSTER-*owe* polecenia co sprowadza go z powrotem do rozsądnych rozmiarów. To zachowanie można zmienić, opcja `--biglog` zabrania wycinania *śmieci*, opcja `--logto=`*plik* kieruje log do konkretnego pliku.

W pliku TEX-owym przeznaczonym do pracy z LOBSTER-em powinna się w pewnym miejscu znaleźć linia:

```
\input lobster
```

Polecenie to nie tylko dostarcza nam pożytecznych makr, ale takze inicjuje czyhający w tle program perlowy, który od tego momentu baczniej zaczyna przyglądać się plikowi `\jobname.log`.

LOBSTER na dzień dobry wzbogaca nas o kilka makr, a ścislej pseudomakr bardzo trudnych do zaimplementowania w czystym TEX-u. Jednym z nich jest `\lbseval #1`, obliczające wartość wyrażenia `#1`, tak naprawdę makro owo nic nie oblicza, jedyne co robi to przekazuje LOBSTER-owi wyrażenie, LOBSTER zaś zwraca wartość w pliku `\jobname.lob.res`, który jest następnie czytany. Wartość przekazywana jest w rejestrze żetonowym `\lbsresult`.

Przykład: wynikiem programu:

```
\newcount \jeden   \newcount \dwa
\jeden = 1  \dwa = 2
\lbseval{
 sin(\the\jeden)*cos(\the\dwa)
}
\message{Wynik=\the\lbsresult}
```

powinno być: `Wynik=-0.350175`

Innym makrem z tej samej rodziny jest `\lbsshell#1`, które wykonuje linię poleceń zawartą jako swój argument i zwraca resultat w pliku `\jobname.lob.res`. Ta operacja nie zwraca wartości w `\lbsresult` ponieważ wynik działania shella może być duży, większy od maksymalnego rozmiaru rejestru żetonowego.

Przykład: Wywołanie `\bsshell{date}\input \jobname.lob.res` wygeneruje napis

```
Sat Jun 1 00:09:35 MET DST 1996
```

Innym przykładem może być polecenie `\lbsshell{tex innyplik.tex}`. Tym razem poprosiliśmy LOBSTER-a o wywołanie następnego zadania TEX-owego, bez kończenia obecnego. Może być nam to potrzebne na przykład do wygenerowania odpowiednich plików `.aux`. Uwaga: LOBSTER bezkrytycznie wykona również polecenia typu: `\lbsshell{reboot}` lub co gorsza `\lbsshell {instaluj-wirusa}`, nie należy więc uruchamiać niesprawdzonych plików pochodzących od osób trzecich.

Budowa modułowa

LOBSTER umożliwia dodawanie do istniejącego systemu nowych pakietów. Każdy pakiet składa się z pliku makr TEX-owych i odpowiadającego im pakietu perlowego. Polecenie `\lbspackage{nazwa}` powoduje przeczytanie pliku `nazwa.lob` i polecenie załadowania do LOBSTER-a pakietu perlowego `nazwa.pm`. Pakiet perlowy czyli plik zaczynający się perlowym poleceniem

```
package Nazwa;
```

ma własności podobne do modułu w języku Modula-2 lub klasy w językach obiektowych. Wszystkie identyfikatory są lokalne dla pakietu i niewidoczne w module głównym i w innych pakietach.

Jednym ze stale rozwijanych pakietow jest `rebox`, pakiet ten dekomponuje TEX-owe pudełko przedstawiając je w postaci listy. Poniższy krótki tekst:

```
\lbspackage{rebox}
\lbsrebox \vbox{
Pierwszy akapit.

Kolejny akapit.
}
```

wygeneruje następujący plik `\jobname.lob.res`:

```
\lbsVbox {1}{1}(18.94444pt+1.94444pt*469.75499pt){%
 \lbsHbox {2}{1}(6.94444pt+1.94444pt*469.75499pt){%
  \lbsHbox {3}{1}(0.0pt+0.0pt*20.0pt){%
  }%
  \lbsFont tenrm\relax
  Pie%
  \lbsKern 0.04167pt\relax
  rwszy%
  \lbsHskip 3.33333pt plus 1.66666pt minus 1.11111pt\relax
  ak%
  \lbsKern -0.05557pt\relax
  apit.%
  \lbsPenalty 10000\relax
  \lbsParfillskip 0.0pt plus 1.0fil\relax
  \lbsRightskip 0.0pt\relax
 }%
 \lbsParskip 0.0pt plus 1.0pt\relax
 \lbsBaselineskip 3.11111pt\relax
 \lbsHbox {2}{2}(6.94444pt+1.94444pt*469.75499pt){%
```

```
20. \lbsHbox {3}{1}(0.0pt+0.0pt*20.0pt){%
21. }%
22. \lbsFont tenrm\relax
23. Ko%
24. \lbsKern -0.11111pt\relax
25. lejn
26. \lbsKern -0.277779pt\relax
27. y
28. \lbsHskip 3.33333pt plus 1.66666pt ↙
       minus 1.11111pt\relax
29. ak%
30. \lbsKern -0.05557pt\relax
31. apit.%
32. \lbsPenalty 10000\relax
33. \lbsParfillskip 0.0pt plus 1.0fil\relax
34. \lbsRightskip 0.0pt\relax
35. }%
36. }%
```

Ukośna strzałka (↙) oznacza, że kolumna biuletynu okazała się za wąska pomimo zmniejszonego stopnia pisma. Następna linijka jest przedłużeniem linijki ze strzałką, należy pominąc w niej początkowe spacje.

Powyższy kod straszliwy powstał w wyniku przerobienia wyniku `\showboxdepth=\maxdimen` `\showboxbreadth=\maxdimen\showbox` i przerobienia perlowym modułem `rebox.pm` zawartości pliku transkrypcyjnego.

Wszystkie polecenia prymitywne zostały zastąpione ich odpowiednikami o nazwach z przedrostkiem `\lbs` i nieco zmienionych parametrach. Polecenia `\lbs`$\frac{\mathrm{H}}{\mathrm{V}}$`box` mają definicję:

```
\def \lbsVbox#1#2(#3+#4*#5){...,
```

gdzie

- `#1` oznacza stopień zagłębienia pudełka. Pierwsze pudełko otrzymuje stopień 1. Pudełko stopnia $n+1$ jest bezpośrednio zagnieżdżone w pudełku stopnia n.
- `#2` oznacza numer kolejny pudełka, zagłebienia są brane pod uwagę. Innymi słowy w numeracji nie jest brana pod uwagę obecność podpudełek.
- `#3`, `#4`, `#5` oznaczają odpowiednio wysokość, głebokość i szerokość pudełka.

Zapis `(#3+#4*#5)` jest nieco mylący, *powierzchnia* pudełka oczywiście wyraża się wzorem $(\#3+\#4)\times\#5$ i tak została prawidłowo umieszczona w pliku transkrypcyjnym, jednak zapis `(#3+#4*#5)` jest nieco łatwiejszy do przeczytania (lub pominięcia).

Zauważmy, że nie zawsze rozmiary pudełka zgadzają się z jego zawartością. Przykładem niech będą pudełka w liniach 3–4 i 20–21, powstałe jako niejawne `\indent` wykonywane na początku każdego akapitu. Obecna wersja programu nie umie sobie z tym problemem poradzić.

Z pudełkiem można związać pewne makro. Polecenie `\lbsOnBox#1#2` dodaje makro przed wykonaniem pudełka o poziomie `#1` i numerze `#2`. Przykładowo: `\lbsOnBox{1}{3}{\vskip 4pt}` dodaje nam 4pt przed trzecim pudełkiem. Można w ten sposób sprobować zaimplementować polecenie `\vadjustafter` działające jak `\vadjust` na n-tą linie akapitu. Wystarczy zamknąć akapit w pudełku, wykonać na nim `\lbsrebox` i zdefiniowac `\lbsOnBox{1}{`n`}{`*materiał pionowy*`}`.

Polecenie `\showbox` ma pewną przykrą cechę. Długość TEX-owego bufora używanego przez operację `\showbox` jest bardzo krótka i nie starcza, jeśli w pudełku występuje żeton `\special` z długą listą parametrów. Przykładowo, krótki programik:

```
\input psfig.sty
\setbox 0 =\hbox{%
\psfig{file=/usr/lib/ghostscript/↙
     examples/tiger.ps, angle=30}
}
\showbox0
\end
```

Przeczytajmy teraz plik transkrypcyjny, znajduje się tam linia:

```
..\special{ps::[begin] 49694912 50224627 ↙
    -23018718 10464880 26676194 60689507 s
t\ETC.}
```

Linia `\special` była tak długa, ze TEX przyciął ją do rozmiarów wystarczających być może do oryginalnego zastosowania polecenia `\showbox`, ale za krótkich do naszych celów. Na tę dolegliwość są dwa lekarstwa. 1) Można przedefiniować polecenie `\special`, tak by numerowało kolejne wywołania, na przykład:

```
\let \@special = \special
\newcount \special@cnt \special@cnt = 0
\def\special {\global \advance \special@cnt
\@special{ps: (special no. ↙
    \the\special@cnt) pop}%
    \@special
}
```

i czytać jednocześnie `.log` i `.dvi`, albo zmienić tak kod TEX-a, żeby długość bufora była większa. Pierwsze rozwiązanie nie jest najszczęśliwsze, drugie z kolei wymaga ingerencji w święty kod TEX-a.

Białe mięso homara czyli Lobster od środka

Program LOBSTER po starcie, zainicjowaniu zmiennych i zainincjowaniu kolejki `\jobsname.log` wykonuje:

```
if((fork()) {
 exec 'tex', "&$format_name", $jobname
}
```

Funkcja systemowa fork(), rozdziela proces na dwa identyczne, w procesie potomnym zwraca 0, utożsamiane z wartością boolowską false, w procesie macierzystym zwraca numer procesu. W procesie macierzystym, (niezerowa wartość fork()), wykonywana jest funkcja systemowa exec, która wykonuje program bedący jej argumentem i... nigdy nie wraca. Drugi proces będący w tym czasie *właściwym* LOBSTER-em spędza większość czasu na czytaniu kolejki, do której TEX, w najlepszej wierze zapisuje plik transkrypcyjny.

Polecenie \input lobster oprócz oczywistej funkcji zdefiniowania kilkunastu makrodefinicji, zapisuje do logu linię:

```
lbs-job: init
end-job
```

Jest to swoisty *self-test* TEX-owej części LOBSTER-a sprawdzający czy na pewno jest słuchany przez Perla. wszystkie polecenia dla LOBSTER-a mają postać:

lbs-job: *nazwa-funkcji*
arg *nazwa* = *wartość*;
⟨*inne argumenty*⟩
end-job

gdzie *nazwa* może być dowolnym identyfikatorem perlowym, wraz z początkowym znaczkiem ($, @ lub %) oznaczającym typ. Brak znaczka jest traktowany jak $ (skalar). Wartość może występować w kilku postaciach:

- Liczba, ten punkt nie wymaga komentarza
- Napis. Napisem jest wszystko co jest poprawnym napisem w sensie Perla. "Ala ma kaca", 'Ala ma kaca', q{Ala ma kaca}, qq<Ala ma kaca> są rownoważnymi napisami. Nazwa moze zawierać koniec linii, no chyba, że wyspecyfikujemy flagę --one-line-strings
- Napis ze znacznikiem końca, wartość w postaci <<KONIEC oznacza, czytaj jako napis następne linijki aż do magicznego słowa KONIEC.
- self-scan, magiczne to słowo oznacza, że procedura perlowa sama przeczyta log i będzie czytała tyle ile zapragnie.

Przed wywołaniem dowolnego zadania należy poprosić LOBSTER-a o dołączenie interesującego nas modułu. Polecenie \lbspackage{rebox} oprócz odpowiedniego \inputa pisze do logu:

```
lbs-job: usepackage
arg name=rebox
end-job
```

Przy implementacji zadań LOBSTER-owych należy zwrócić uwagę na następujące czynniki opóźniające wykonywanie polecenia zewnętrznego:

- TEX ma dosć pokaźnych rozmiarów bufor dla pliku transkrypcyjnego.
- Polecenie zewnętrzne wykonuje się asynchronicznie i TEX czekając dla wykonanie może albo zająć się innym zadaniem, albo co bardziej prawdopodobne czeka w nieskończonej pętli zajmując cenny czas procesora(ów).

Z pierwszym problemem można sobie poradzić sztucznie przepełniając bufor. Linia

```
lbs-fill: Jakiś dłuuuuuuuugi tekst.
```

dopisana po end-job jest pomijana w procesie czytania pliku i służy wyłącznie do przepełniania bufora. TEX w momencie oczekiwania na wykonanie zadania wpisuje w pętli lbs-fill, zapewniając jednocześnie przepełnienie bufora i co za tym idzie przeczytanie zadania przez LOBSTER-a oraz wywołanie sztucznego *korka* w kolejce, albowiem LOBSTER w chwili wykonywania zadania kolejki nie czyta. W środowiskach *unixowatych* proces, którego dane nie zostały przeczytane potokowo jest usypiany aż do momentu uwolnienia kolejki.

Wnioski

Wydaje się, ze TEX jako system może być (jeśli to w ogóle możliwe) jeszcze ciekawszy po dodaniu mu możliwości korzystania z otoczenia bez wychodzenia z aktualnego zadania. W ten sposób można sprobować rozwiązać zadania, które wymagałyby bardzo wielu przebiegów.

Program LOBSTER jest obecnie w bardzo wczesnym stadium rozwoju, pomimo, że już pracuje jest jeszcze ciągle właściwie w fazie projektowania. Postaram się, żeby wersje β programu ujrzały jak najszybciej swiatło dzienne.

◇ Krzysio Leszczyński
chris@camk.edu.pl