Wejście/Wyjście w XEmacsie

Krzysztof Leszczyński
Warszawa
chris@camk.edu.pl

Streszczenie

Emacs to duży system, można w nim wiele oprogramować. Jak w każdym dużym systemie, na nieostrożnego programistę czeka wiele pułapek.

Wprowadzenie

(X)Emacs jest dużym systemem i dosyć dużo umie. Ponadto jest bardzo programowalny. W zasadzie nie można go uważać za edytor tekstu z makrodefinicjami, ale raczej za potężne środowisko, w którym edycja tekstów jest jednym z zastosowań.

Jest to szczególnie przydatne w przypadku konieczności edycji tekstów, których nie da się w sposób standardowy zapisać. Jeśli potrzebujemy w jednym dokumencie zapisać tekst polski i rosyjski, to musimy użyć jednocześnie dwóch kodowań.

Emacs występuje w wielu odmianach, dwie najbardziej znane to oryginalny GNU Emacs oraz `XEmacs`. Ponadto każda z nich może być skompilowana ośmiobitowo lub w wersji MULE (Multilingual extension of Emacs). W wersji MULE emacs przechowuje dane w postaci niekoniecznie jednobajtowej – na ogół jako ISO-2022 lub Unicode.

Niestety, wersja `XEmacs`-a, której używam do pisania tego artykułu – 21.1.14 (Cuyahoga Valley) nie potrafi, w wersji MULE, prawidłowo obsłużyć tak potrzebnych funkcji jak `upcase-word`. Funkcja ta działa prawidłowo w GNU Emacs. Jeśli chcę używać `XEmacs`-a, muszę, na razie, stosować kodowanie ośmiobitowe. To się prawdopodobnie niedługo zmieni. Z chwilą gdy to nastąpi, będzie można diametralnie zmienić podejście i używać niemal wyłącznie Unicode do jednolitej reprezentacji wewnętrznej i niemal dowolnego kodowania w plikach.

GNU Emacs i `XEmacs` są do siebie bardzo podobne. Bardzo często kod napisany na jedną odmianę będzie poprawnie pracował na innych, jednak różnice pomiędzy nimi są znaczne. Niestety podstawową różnicą jest definicja zdarzenia, w `XEmacs`-ie zdarzenie stanowi oddzielny typ. Funkcje manipulujące zdarzeniami różnią się od siebie, dlatego niezwykle ciężko jest stworzyć kod, który byłby przenośny. W artykule opisuję zdarzenia w `XEmacs`-ie.

Zdarzenia

`XEmacs` większość swojego życia spędza w pętli:

- sprawdź czy w kolejce jest jakieś zdarzenie (*ang. event*) do obsłużenia,
- obsłuż zdarzenie,
- uaktualnij wygląd (a zwłaszcza treść) okien.

Różne systemy operacyjne i okienkowe przekazują różne zdarzenia. Podczas pracy na konsoli tekstowej, praktycznie jedynymi zdarzeniami będą te zgłaszane przez klawiaturę, a ściślej terminal oraz sygnały wysyłane przez system operacyjny. W systemach okienkowo-graficznych (X11, MS-Windows, MacOS) zdarzeń będzie odpowiednio więcej, gdyż środowisko jest bardziej skomplikowane. `XEmacs` rozpoznaje kilka rodzajów zdarzeń, nas będą głównie interesować zdarzenia związane z klawiaturą – `key-press`, oraz w nieco mniejszym stopniu z przyciskami myszy.

`XEmacs` ukrywa przed programistą szczegóły systemu operacyjnego. Dzięki temu, pliki lispowe nie muszą być implementowane oddzielnie dla każdego środowiska. Funkcja `(next-event)` czeka na kolejne zdarzenie i zwraca je, usuwając z kolejki zdarzeń. Tę funkcję wykorzystujemy bardzo rzadko, gdyż zwraca nam wszystkie zdarzenia, o których prawdopodobnie nie chcemy słyszeć – ruchy myszy, zmiana rozmiarów okna, itd. Klawisze *czytamy* funkcją `(next-command-event)`, która zwraca nam wszystkie *w zasadzie interesujące* zdarzenia, prawidłowo obsługując w tle, wszystkie pozostałe.

Zdarzenia klawiszowe

Zdarzenia, które opisują odpowiednie klawisze są wektorami nazw, bądź list w postaci `(`*modyfikatory klawisz*`)`. W dalszej części będę używał zamiennie terminów *zdarzenie* i *klawisz*, chyba, że będzie potrzebne rozróżnienie tych terminów. `XEmacs` nie odróżnia, w zasadzie, klawiszy myszy od klawiszy klawiatury. Obsługę obu zdarzeń definiujemy za pomocą funkcji `define-key`. Gdybym chciał opisać dokładnie działanie tej funkcji, gościnne łamy biuletynu GUST nie pomieściłyby nawet tłumaczenia opisu tej funkcji w podręczniku `XEmacs`-a. Dlatego podam wersję uproszczoną, zaznaczając, że jest to tylko część prawdy. Dokładniejszy opis tej funkcji można znaleźć w XEmacs-owym systemie pomocy oraz w dokumencie XEmacs *Lisp Reference Manual*, dostępnym w pakiecie XEmacs. Funkcja define-key ma trzy argumenty:

(define-key *mapa*
sekwencja-zdarzeń
definicja)

mapa jest obiektem typu keymap. Obiekty takie możemy uważać za tablice asocjacyjne, w którym kluczami są zdarzenia typu command-event. Argument *sekwencja-zdarzeń* składa się z jednego lub większej ilości zdarzeń w postaci:

- znaku lub liczby naturalnej, tylko znaki o reprezentacji ASCII mogą być, w sposób przenośny, reprezentowane w ten sposób. Warto pamiętać, że znaki o kodach większych niż 126 nie mogą być reprezentowane w ten sposób, gdyż starsze wersje Emacs-ów używały konkretnych bitów do zapisywania modyfikatorów. Generalnie należy unikać takiego zapisu i stosować zapis symboliczny.
- symbol, na przykład 'a lub 'aogonek, oznacza klawisz o takiej nazwie. Jeśli pracujemy w środowisku, w którym litery są zapisywane w kodowaniu ASCII to *znak* 65 i znak 'A będą zapisami równoważnymi, zaś w mapie zostanie zapisany klucz 'A.
- lista w postaci (*modyfikatory klawisz*), gdzie modyfikator jest jednym z symbolów: 'shift, meta, alt. '(meta control f7) oznacza klawisz f7 z wciśniętymi jednocześnie klawiszami meta i control. W przypadku klawiatur PC, meta oznacza Alt. Lista modyfikatorów może być pusta. Lista w postaci '(161) oznacza znak o kodzie 161, w odróżnieniu od, wydawałoby się oczywistego zapisu 161, który oznacza znak o kodzie $161 - 128 = 33$ czyli „!" z modyfikatorem 128 (meta). Wszędzie tam, gdzie mogą powstać wątpliwości co do sposobu interpretacji, należy stosować pełny zapis listowy.
- Wektor, złożony z wyrażeń wymienionych powyżej. [(control x) (control f)] oznacza klawisz „x" z klawiszem *control*, po którym następuje klawisz „f", również z klawiszem *control*.

Ostatnim argumentem jest *definicja*, która może mieć jedną z postaci:

- nil, oznacza, że klucz ma być usunięty z mapy,
- [w (control e) k t o (r)]. Ten zapis możemy traktować jak makrodefinicję. Do kolejki zdarzeń zostaną dołożone zdarzenia będące elementami wektora. Zdarzenia te zostaną przetworzone, po jednym, przy okazji następnego wywołania (next-event). W wymienionym przykładzie, przy założeniu, że XEmacs posługuje się domyślną mapą klawiatury zostanie wykonane operacje: w – czyli dopisanie do bufora znaku w, przejście na koniec wiersza (C-e) oraz dopisanie liter k t o r. Ostatnia litera jest napisana jawnie, jako lista bez modyfikatora.
- "napis", jest to skrócona forma zapisu wektorowego, XEmacs wykonuje niejawnie operację (string-to-vector "napis"), uwaga: napis jest rozkładany na klawisze bez modyfikatorów, czyli podlega staroemacsowej siedmiobitowej tradycji. Gdybyśmy napisali
  (define-key global-map [(f7)] "żółć"),
  to poza faktem, że żółć by nas mogła zalać, nie uzyskalibyśmy zamierzonego efektu w buforze. ?ż, czyli w kodowaniu ISO-8859-2, znak o kodzie 191 oznacza znak o kodzie $191 - 128 = 63$, czyli poczciwy znak zapytania, z wciśniętym modyfikatorem meta. Standardowo taki znak nie jest w XEmacs-ie zdefiniowany – XEmacs zgłosi błąd.
- Obiekt typu keymap, oznacza zinterpretowanie następnego zdarzenia, zgodnie z tą mapą. Mapy tak naprawdę nie potrafią przechowywać ciągu zdarzeń, tylko pojedyncze zdarzenie, jeśli podamy funkcji define-key ciąg, to funkcja skonstruuje ciąg odpowiednich map.
- Lista typu (*mapa klawisz*). XEmacs przypisze zdarzeniu definicję: weź definicję dla klawisza będącego cadr-em listy z mapy będącej car-em listy.
- (lambda () "*opcjonalna dokumentacja*"
  (interactive) *treść*), naciśnięcie klawisza spowoduje, że λ-wyrażenie zostanie wykonane. Wyrażenie musi być *interaktywne*, czyli zawierać jak swój pierwszy element wywołanie funkcji (interactive). W przypadku map key-translation-map lub function-key-map odpowiednie lambdy nie muszą być interaktywne.
- 'symbol, do klawisza będzie przypisana funkcja o tej nazwie.

System operacyjny a sprawa polska

Chcielibyśmy, aby programy XEmacs-owe czytały klawiaturę w sposób przenośny, to znaczy, żeby klawisze nazywały się – tam gdzie to możliwe tak samo, niezależnie od systemu operacyjnego i środowiska. Nie ma większych problemów z klawiszami

ASCII, jest mało prawdopodobne, żeby XEmacs nie przetłumaczył znaku o kodzie 65 na symbol 'A. Jak zwykle, Słowianie mają gorzej. Nad kodowaniem polskich liter wylano już morze łez i atramentu, truizmem będzie zatem stwierdzić, że różne systemy będą generowały polskie znaki różnie. Jedno z bardziej ciekawych podejść przedstawia tutaj środowisko X11. Od dawna (a pamięcią sięgam około 13 lat, co jest dawno w sensie systemów okienkowych) X-Window przekazywały znaki w postaci symbolicznej. Było zatem bardzo łatwo dodać znak aogonek – udało się to za pierwszym razem na maszynie Sun-3, pochodzącej z wczesnych lat 80-tych. XEmacs rozumie owe symbole bezpośrednio, funkcja (next-command-event) zwróci symbol 'ogonek. Oczywiście gdyby wszystko zrobione tak do końca, to byłoby za dobrze – dlatego Ł nie jest przekazywane jako Lslash tylko, w zależności od konkretnej wersji systemu, jako Lstroke lub Lstroked.

Inne systemy przekazują znaki, na ogół w postaci numerycznej, z czym musimy sobie też poradzić. Dobrym pomysłem jest tłumaczenie, na samym wstępie, podczas czytania zdarzenia znaku lub sekwencji znaków na odpowiednie symbole. Po przetłumaczeniu możemy zapomnieć o związku znaku z konkretną klawiaturą i środowiskiem. Żeby to zrobić efektywnie, musimy prześledzić drogę jaką przebywa zdarzenie w środowisku lispowym XEmacs-a. Nieco wulgaryzując, można powiedzieć, że XEmacs spędza niemal cały czas w pętli

```
(while (let ((zdarzenie (next-event)))
         (dispatch-event zdarzenie)))
```

zaś funkcja dispatch-event, niczym Kim Ir Sen, wie wszystko co trzeba zrobić z konkretnym zdarzeniem, niezależnie od tego czy zdarzenie dotyczy wciśnięcia klawisza czy zmiany palety kolorów. W przypadku interesujących nas zdarzeń, funkcja *przepuszcza* je przez szereg map:

1. function-key-map, przy czym wpis w tej mapie jest ignorowany, o ile występuje w mapach, wymienionych dalej. Ta mapa jest w zasadzie używana wyłącznie przez autorów XEmacs-a. Jest inicjowana na bardzo wczesnym etapie. Jej zawartość jest silnie zależna od systemu operacyjnego i środowiska. Mapa służy do zamiany symboli wysyłanych przez system na symbole bardziej zrozumiałe dla pozostałej części kodu emacsowego. Na przykład dla terminala VT100 mapa zawiera definicję ciągu ESC [[A jako klawisza (symbolu) f1.
2. key-translation-map, ta mapa jest przeszukiwana przed dowolną inną mapą (z wyjątkiem systemowej function-key-map).

Te dwie mapy nie są typowe. Wartością funkcji powinien być ciąg (wektor) znaków, na które odpowiednie klawisze są zamieniane, ponadto funkcje mogą przyjmować jeden argument napis, który się pojawia, jeśli przez odpowiednio długi czas, około 1 sekundy, nie naciskamy żadnego klawisza w czasie wykonywania funkcji next-command-event.

Dalej, dla zdarzeń klawiszowych (w odróżnieniu od myszatych) przeszukiwane są mapy:

3. mapa, będąca wartością własności keymap obszaru w którym jest aktualnie (point).
4. kolejno wszystkie mapy związane z trybami pomocniczymi (*ang. minor modes*).
5. mapa lokalna związana z buforem.
6. global-map

Dla zdarzeń generowanych przez mysz, przeszukiwanych jest około 2 razy więcej różnych map. Omawianie ich wykracza poza ramy tego artykułu.

Spróbujmy zaimplementować notację ciachową. W pierwszym podejściu próbujemy tylko literę „ą":

```
(define-key "/a" "ą")
```

To się oczywiście nie uda, gdyż podstawową definicją dla klawisza „/" (i nazwie '/, gdyż jak wiadomo *ciach* jest bardzo dobrą nazwą w języku Lisp) jest funkcja self-insert-command, która powoduje umieszczenie odpowiedniego znaku w odpowiednim miejscu bufora. Większość *zwykłych* klawiszy ma tę funkcję przypisaną – w mapie global-map. Zanim zatem przypiszemy "/a" lepiej jest przypisać znakowi ?/ definicję nil, dzięki czemu XEmacs nie będzie protestował przeciwko niejednoznacznemu przypisaniu jednocześnie do ?/ i do "/a".

Generalnie, nie jest dobrze zmieniać zbyt dużo w global-map, zrobimy zatem inaczej. Do global-map dopiszemy definicję dla aogonek:

```
(define-key [(aogonek)] [(?ą)]
```

a następnie nauczymy XEmacs-a, że aogonek uzyskujemy (również) poprzez *ciach*-a.

Zdefiniujmy tryb pomocniczy PL-ciach-mode, robimy to następująco

```
(defvar PL-ciach-mode nil
  "Non-nil causes "/" to interpreted
   as a prefix key")
(make-variable-buffer-local
 'PL-ciach-mode)
```

oraz pomocniczą listę:

```
(defconst
  PL-keyboard-translations
  '( (?a . aogonek) (?A . Aogonek)
     (?c . cacute)  (?C . Cacute)
     (?e . eogonek) (?E . Eogonek)
     (?l . lslash)  (?L . Lslash)
```

```
        (?n . nacute)  (?N . Nacute)
        (?o . oacute)  (?O . Oacute)
        (?s . sacute)  (?S . Sacute)
        (?x . zacute)  (?X . Zacute)
        (?z . zdotaccent)
        (?Z . Zdotaccent))
     "The prefixing notation consists of
      a prefix character, usually
      ?/ and a letter.")
```

Do mapy `key-translation-map` dopisujemy tylko ten znak, który posłuży nam jako *ciach*, czyli, zgodnie ze starą indiańską tradycją wektor złożony z jednoelementowej list zawierającej symbol ,,/'':

```
(define-key key-translation-map [(/)]
   'PL-check-ciach)
```

zaś samą funkcję definiujemy:

```
(defun PL-check-ciach (prompt)
 "Sprawdź czy po ciachu jest właściwy
  klawisz"
```

Jeśli nie jesteśmy w trybie `PL-ciach-mode`, zwróć ten samo zdarzenie, które nie zostanie ponownie przetworzone przez `key-translation-map`.

```
(if (not PL-ciach-mode)
    [ (/) ]
```

W przeciwnym wypadku sprawdźmy czy następne zdarzenie nie jest przypadkiem, interesującym nas znakiem

```
(let* ((next-key
        (event-key
         (next-command-event nil)))
```

Sprawdzamy (cały czas wewnątrz konstruktora symboli lokalnych `let*` czy znak nie jest w tablicy:

```
(ksym
 (cdr
  (assq next-key
 PL-keyboard-translations))))
```

Teraz możemy już zacząć zwracać wynik. Porządny wynik powinien być wektorem list, żeby `XEmacs` nie miał żadnych wątpliwości jak ma go zinterpretować:

```
(vector
 (cond
```

Jeżeli nasz znak został znaleziony w liście asocjacyjnej `PL-keyboard-translations`, to po prostu go zwracamy

```
(ksym (list ksym))
```

Jeśli nie, to sprawdźmy, czy nie było dwóch *ciach*-ów, jeśli były to zwracamy pojedynczy *ciach*:

```
((eq next-key ?/)
     '(?/))
```

Dawno temu, pewien GUST-owiec imieniem Stanisław ustalił, że *ciach-spacja* powinno generować ciach, i tak już zostało, pytanie tylko czymże jest spacja. Znakiem `' '`, kodem 32, czy symbolem `'space`? Sprawdzimy wszystkie możliwości:

```
((or (eq next-key ?\ )
     (eq next-key 32 )
     (eq next-key 'space))
    '(?/))
```

W pozostałych przypadkach też zwrócimy znak `/`, ale dodatkowo wrzucimy do kolejki drugi ze znaków, w ten sposób wciśnięcie `/q` spowoduje dopisanie do bufora `/q`:

```
(t (setq unread-command-events
    (cons
     (make-event 'key-press
      (list 'key next-key))
     unread-command-events))
    '(/)  )))))))
```

Teraz notacja ciachowa działa dla wszystkich buforów, które mają włączony `PL-ciach-mode`. Jednocześnie, o ile nasze środowisko generuje poprawnie znaki polskie jako symbole, np. jako `aogonek` to oba sposoby wprowadzania są równoważne.

Pytanie, dlaczego nie można „/” zdefiniować w lokalnej mapie trybu `PL-ciach-mode`. Odpowiedź nie jest prosta. W zwykłym wprowadzaniu znaków rzeczywiście wystarczy, niemniej tryby specjalne, w tym tryb `I-Search` czytają bezpośrednio kolejkę zdarzeń, dlatego o ile chcemy mieć polskie znaki także w trybie `I-Search` należy podmiany dokonać odpowiednio wcześnie.

Twarde spacje

Jako kolejną zastosowanie map zdefiniujmy tryb, w którym klawisz *spacja* wstawia tyldę o ile następuje po pewnych słowach. Tryb definiujemy:

```
(defvar PL-hardspace-mode nil
  "See PL-hardspace-mode function")
```

Oczywiście nie chcemy mieć tyld w każdym buforze, tylko w wybranych.

```
(make-variable-buffer-local
 'PL-hardspace-mode)
```

Zdefiniujmy napis, złożony z listy słów po których nie chcemy dzielić, rozdzielonych spacjami. Zdefiniujmy go jako `defcustom`, żeby użytkownik mógł sobie zmieniać wartości, nie ruszając źródeł samego programu.

```
(defcustom PL-hardspace-words
  "na z o pod i a u się od do w po dr \
prof. mgr doc."
```

```
    "The list of Polish words after which
  there should be a hard space rather than
  a soft. If you change this variable
  reinvoke  PL-hardspace-mode to recompute
  regular expressions."
    :type '(string)
    :group 'ciach)
```

Z listy będziemy tworzyć wyrażenie regułowe, na razie zadeklarujmy zmienną

```
  (defvar PL-hardspace-regexp nil
    "A regular expression used to decide
  when emacs should trigger  a hardspace.")
```

Wejście w tryb oznacza ustawienie, w zależności od argumentu funkcji, zmiennej `PL-ciach-mode` oraz obliczenie wyrażenia regułowego na podstawie napisu `PL-hardspace-words`.

```
  (defun PL-hardspace-mode (&optional arg)
    (interactive "P")
    (setq PL-hardspace-mode
          (if (null arg)
              (not PL-hardspace-mode)
            (> (prefix-numeric-value arg)
               0)))
```

Przygotujemy reguły: dzielimy łańcuch według spacji i z otrzymanej list budujemy napis `\W\(`*lista napisów, rozdzielona znakami* `\|\)`.

```
    (setq PL-hardspace-regexp
     (concat "\\W\\("
      (substring
       (apply
        'concat
         (mapcar
          (lambda (a)
           (concat "\\|"
                   (regexp-quote a)))
          (split-string
           PL-hardspace-words)))
        2) "\\)")))
```

Teraz definiujemy mapę klawiatury, która ma tylko jeden wpis – dla spacji. `(make-sparse-keymap)` konstruuje mapę, która jest zoptymalizowana do obsługi niewielkiej ilości wpisów. W tym wypadku tylko jednego.

```
  (defvar PL-hardspace-mode-map
    (let ((kmap (make-sparse-keymap)))
      (define-key kmap [space]
                  'PL-try-space)
     kmap)
    "A keymap for dynamic space, that
  is sometimes <hard>")
```

Dodajemy tryb `PL-hardspace-words` do listy trybów, jeśli tryb będzie włączony to automatycznie będzie użyta mapa `PL-hardspace-mode-map`. Jako nazwę trybu wybierzemy `~`, co nam nie powinno zaśmiecić dolnej części ekranu.

```
    (add-minor-mode 'PL-hardspace-mode "~"
     PL-hardspace-mode-map)
```

A oto sama funkcja

```
    (defun PL-try-space () (interactive)
      "Check whether insert a space or a
    tilde or other non-breakable thing"
```

Jeśli ostatnio w buforze została wprowadzona tylda, następna spacja zamienia tyldę na spację, a konkretnie na znak, który ostatnio wciśnięto.

```
      (if (equal (buffer-substring
                  (max 1 (- (point) 1))
                  (point))
                 "~")
          (progn (delete-backward-char 1)
                 (self-insert-command 1))
```

W przeciwnym wypadku, sprawdź czy ostatni wyraz spełnia wyrażenie regułowe, jeśli tak wstaw tyldę, jeśli nie, pozwólmy wykonać całą robotę funkcji `self-insert-command`.

```
        (let ((p (point)))
          (if (re-search-backward
               PL-hardspace-regexp
               (- (point) 10) t)
              (progn
                (goto-char p)
                (if (eq (match-end 1) p)
                    (insert "~")
                  (self-insert-command 1)))
            (self-insert-command 1)))))
```

Wszystkie te funkcje (w nieco rozbudowanej postaci) można znaleźć w pliku `ciach.el` dostępnym pod adresem:
`ftp://ftp.camk.edu.pl/private/chris/ciach/`

Krzysztof Leszczyński
Warszawa
`chris@camk.edu.pl`

Polska Grupa Użytkowników Systemu TEX, 2001 (http://www.GUST.org.pl)