

Programowanie funkcyjne 2014

Propozycje projektów

(wersja wstępna)

Grupa kzi

Uwagi

- To są propozycje projektów, więc nie trzeba ich robić kropka w kropkę, tak jak jest tu napisane. W szczególności, jeśli przesadziłam gdzieś z wymaganiami, to można je “poluzować”. Z tym, że warto konsultować modyfikacje, bo jeśli projekt okaże się zbyt prosty, to będziemy ciąć punkty.
- Projekty powinny być przenośne. Tzn. powinno się je dać skompilować i na Windows’ie i na Linux’ie. Chyba, że indywidualnie ustalimy inaczej.
- Styl programowania będzie oceniany. Tj. kod powinien być czytelny i mieć logiczną strukturę.
- Projekty raczej nie są jednakowo pracochłonne, ale ani nie mam czasu, żeby je wszystkie zaimplementować, ani wystarczającego doświadczenia, żeby to ocenić na oko. Dlatego, żeby przywrócić trochę równowagi, projekty, które okażą się wyraźnie łatwiejsze od innych będą surowiej oceniane, a projekty trudniejsze na odwrót. Oczywiście nie znaczy to, że z łatwiejszych nie będzie można dostać max’a.
- Wszystkie projekty są równoważnościowe ... ale jeszcze się nie zdecydowałam, ile punktów za nie przyznać. Prawdopodobnie coś w okolicach 100.

A. Downloader Aplikacja ściąająca pliki przez HTTP (przede wszystkim strony).

Wejście: Lista URL'i, opcjonalna lista ścieżek do katalogów i opcjonalna flaga `r`.

Specyfikacja: Aplikacja ściąaga z Internetu pliki wskazane przez URL'e podane na wejściu i umieszcza je we wskazanych katalogach. Jeśli na wejściu nie podano tych katalogów, to aplikacja umieszcza pliki w katalogu bieżącym.

Jeśli URL wskazuje na plik HTML lub CSS, nazwijmy go *f*, i flaga `r` jest ustawiona, to aplikacja rekurencyjnie ściąaga też wszystkie pliki, od których *f* zależy i których URL'e są nie płytsze niż URL podany na wejściu. Przy czym zakładamy, że zależenie jest relacją przechodnią oraz, że adresy występujące jako tekst nie są zależnościami.

Przykład. Jeśli zażądaliśmy ściągnięcie `www.xyz.com/cos_tam/index.html` i mamy w nim link do `www.xyz.com/cos_tam/gallery.html`, a w `gallery.html` mamy umieszczone obrazki `www.xyz.pl/cos_tam/pics/1.jpg`, `www.xyz.pl/cos_tam/pics/pics/2.jpg`, to aplikacja powinna ściągnąć wszystkie te pliki. Ale jeśli w `index.html` występują linki do `www.xyz.com/main.html` i `www.abc.com/cos_tam/sponsor.html`, to pliki `main.html` i `sponsor.html` nie powinny być ściągnięte.

URL'e do ściąganych zależności powinny być podmieniane w kodzie HTML i CSS na URL'e do ich lokalnych kopii.

Krótko mówiąc, dobrze napisana strona nie wykonująca obliczeń po stronie serwera powinna po ściągnięciu z opcją `r` działać i wyglądać w trybie offline, tak jak jej oryginał w sieci.

Poza tym, ściąganie każdej pozycji z listy wejściowej powinno się odbywać w osobnym wątku.

To jest pakiet do HTTP dla Haskell'a:

`http://hackage.haskell.org/package/HTTP/docs/Network-HTTP.html`

Pod Windows'em możliwe, że trzeba będzie przeinstalować pakiety `network` i `HTTP`. Do tego potrzebny jest `MSYS`. `MSYS`'a instaluje się stąd:

`http://www.mingw.org/wiki/Getting_Started`

Przy czym nie zaznaczamy w instalatorze MinGW'a, bo może przysłonić MinGW'a w GHC. Pakiety przeinstalowujemy z konsoli `MSYS`'a za pomocą `cabal`'a (jest instalowany razem z GHC). Jeśli przy instalacji pakietów będą nam się sypać błędy linkera, trzeba w `MSYS`'ie ustawić zmienną środowiskową `MSYSTEM=MINGW32` albo `MSYSTEM=MINGW64` (w zależności od tego jaką wersję GHC zainstalowaliśmy).

Tu jest tutorial do wątków w Haskell'u:

`http://chimera.labs.oreilly.com/books/1230000000929/ch07.html`

Biblioteki dla OCaml'a do HTTP nie są zbyt przenośne, więc jeśli chcemy to pisać w OCaml'u, to powinniśmy zrobić własną obsługę wiadomości HTTP (to nie jest trudne, musimy tylko umieć konstruować zapytanie GET i wyciągać z odpowiedzi serwera interesujące nas dane).

Tu jest tutorial do wątków w OCaml'u:

`http://ocaml.github.io/ocamlunix/threads.html`

Uwaga: Te wątki są ciężkie, więc nie powinniśmy ich stosować w aplikacjach, które tworzą bardzo dużo wątków (podobno dla dzisiejszych PC'tów nie powinno być ich więcej niż 100).

B. Generator lekserów

 Aplikacja generująca lekser z opisu gramatyki regularnej.

Wejście: Ścieżka do pliku z opisem gramatyki regularnej i tokenów, oraz ścieżka do pliku, w którym ma być umieszczony lekser.

Przykładowy plik z gramatyką może wyglądać tak:

```
tokens: number , operator , lparenthesis , rparenthesis , space

number  -> '-' nat | nat
nat     -> '0' nat_aux | '1' nat_aux | ... | '9' nat_aux
nat_aux -> nat | .

operator -> '+' | '-' | '*' | '/'

lparenthesis -> '('
rparenthesis -> ')'

space      -> ' ' space | ' '
```

gdzie reguła postaci $x \rightarrow y \mid \dots \mid z$ to skrót dla listy reguł $x \rightarrow y, \dots, x \rightarrow z$, $.$ to epsilon (ciąg pusty), zwykły napis to symbol nieterminalny, a znak w cudzysłowach, to znak z alfabetu (symbol terminalny).

(Oczywiście było by lepiej, gdybyśmy w regułach mogli używać wyrażeń regularnych.)

Specyfikacja: Aplikacja zapisuje w pliku leksera kod w wybranym języku programowania, który w czasie liniowym dzieli stringi na listy tokenów zgodnie ze zdefiniowaną w pliku gramatyką.

Dokładniej, aplikacja:

- Dla każdego tokena S buduje niedeterministyczny automat A_S skończony z epsilon-przejściami równoważny gramatyce $\langle N, \Sigma, P, S \rangle$, gdzie N to zbiór wszystkich symboli nieterminalnych w pliku, Σ to zbiór znaków ASCII, P to zbiór wszystkich reguł w pliku. (Tu się nic nie dzieje. Te gramatyki odpowiadają tym automatom 1 do 1.)
- Łączy wszystkie automaty z poprzedniego punktu w jeden automat A , który rozpoznaje słowo wtw gdy jakikolwiek automat A_S je akceptuje.
- Usuwa epsilon przejścia z A . Determinizuje A . Minimalizuje A . (Dwa ostatnie są na Wikipedii. Pierwsze jest trywialne.)
- W tym momencie każdy stan automatu A odpowiada zbiorowi stanów początkowego A . Jeśli w zbiorze tych super stanów występuje taki, który odpowiada stanom akceptującym w więcej niż jednym automacie A_S aplikacja kończy się błędem (ponieważ oznacza to, że istnieją napisy, które mogą być zinterpretowane jako dwa różne tokeny).
- Definicje w pliku leksera typ dla tokenów — każda wartość tego typu pamięta rodzaj tokena i string, który został jako ten token zinterpretowany (np. `type token = Number of string | Operator of string | ...`).
- Zapisuje A do pliku leksera w postaci tablicy dwuwymiarowej t w taki sposób, że w A istnieje przejście $q \xrightarrow{a} q'$ wtw $t[q, ASCII_CODE(a)] = q'$.
- Zapisuje do pliku leksera funkcję, która bierze string'a, znajduje najdłuższy jego prefiks akceptowany przez A i zwraca token zawierający ten prefiks oraz pozostały sufiks. Jeśli taki prefiks nie istnieje, funkcja kończy się błędem. Jeśli prefiks został zaakceptowany i wczytywanie go zakończyło się w stanie, który odpowiada stanowi akceptującemu automatu A_S , to rodzajem tokena jest S .

Czyli zaaplikowanie powyższej funkcji wygenerowanej z powyższej gramatyki do napisu "123 +456", powinno zwrócić token `Number("123")` i sufiks " +456".

C. Mini forum Aplikacja typu klient-serwer.

Serwer-wejście: Adres i port, na którym serwer ma nasłuchiwać i ścieżka do katalogu, w którym trzymana jest kopia forum i dane użytkowników.

Serwer-specyfikacja: Serwer łączy się przez TCP. Obsługa każdego połączenia odbywa się na osobnym wątku. Serwer potrafi rejestrować nowych użytkowników, logować, wylogowywać, dodawać wątki i wpisy, wysyłać listę wątków, wysyłać listę n ostatnich wpisów. Po każdej modyfikacji forum jest archiwizowane we wskazanym katalogu i dearchiwizowane (wczytywane) przy uruchomieniu serwera. Dane powinny być zarchiwizowane tak, żeby można było je ręcznie modyfikować.

Klient-wejście: Adres i port serwera, z którym klient ma się połączyć.

Klient-specyfikacja: Klient łączy się z zadany serwerem przez TCP i wykonuje instrukcje wprowadzane przez użytkownika (np. w terminalu). Instrukcje pokrywają się z akcjami, które potrafi wykonywać serwer.

Tu jest tutorial do robienia serwera TCP na wątkach w Haskell'u:

https://www.haskell.org/haskellwiki/Implement_a_chat_server

Tu jest mały klient TCP w Haskell'u:

<https://gist.github.com/1100407/77f43a49bb68bd7817f6fcae6b661275ac19c0d7>

Tu jest tutorial do socket'ów w OCaml'u:

<http://ocaml.github.io/ocamlunix/sockets.html>

Tu jest tutorial do wątków w OCaml'u:

<http://ocaml.github.io/ocamlunix/threads.html>

Uwaga: Te wątki są ciężkie, więc nie powinniśmy ich stosować w aplikacjach, które tworzą bardzo dużo wątków (podobno dla dzisiejszych PC'tów nie powinno być ich więcej niż 100).

D. Tetris Prosta gra w OpenGL'u.

Wejście: Poziom trudności.

Wyjście: Liczba zdobytych punktów.

Specyfikacja: Jakby ktoś nigdy nie grał w Tetris'a, to może go sobie obejrzeć tutaj:

<https://www.freetetris.org/game.php>

Punktacji nie trzeba wyświetlać na bieżąco. Poziom trudności może, na przykład, oznaczać szybkość spadania klocków.

Tutorial do OpenGL'a w Haskell'u można znaleźć tu:

<https://www.haskell.org/haskellwiki/OpenGLTutorial1>

<https://www.haskell.org/haskellwiki/OpenGLTutorial2>

Tu są jakieś przykłady do OpenGL'a w OCaml'u:

<https://www.peterkrantz.com/2008/hello-opengl-world-in-ocaml/>

<http://www.ffconsultancy.com/ocaml/maze/index.html>

<http://www.ffconsultancy.com/ocaml/mandelbrot/index.html>

Tu jest biblioteka do OpenGL'a dla OCaml'a (mi udało się ją skompilować tylko do wirtualnej maszyny, ale do Tetris'a to powinno wystarczyć):

<http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lab1gl.html>