Narzędzia

✓

Po funkcjach do TEX-a

Ryszard Kubiak

Użytkownicy TEX-a często napotykają sytuacje, gdy trzeba wstępnie przetworzyć dane, zanim zostaną one przekazane do TEX-a. Zdarzyć się to może, gdy mamy do dyspozycji dane liczbowe, będące wynikiem pracy urządzenia lub programu, ale ich postać jest niewygodna do obróbki wewnątrz TEX-a. Niekiedy wyniki działania samego TEX-a, dajmy na to wygenerowane skorowidze, wymagają obróbki, na przykład przesortowania, zanim zostaną włączone do składu.

Utarła się praktyka, aby do wstępnej obróbki danych, zwłaszcza tekstowych, wykorzystywać języki skryptowe typu AWK lub PERL. Zwłaszcza AWK, z racji zwięzłości notacji, sprawności wykonywania programów oraz faktu istnienia implementacji dla większości platform systemowych, zyskał sporą popularność. Ma się rozumieć, że każdy używa takich narzędzi, jakie ma dobrze opanowane i które dobrze spełniają jego oczekiwania. Autor niniejszego artykułu nabrał doświadczenia w wykorzystywaniu języka funkcyjnego Haskell do przygotowywania danych dla TEX-a/LATEX-a.

Haskell należy do rodziny języków *funkcyjnych*. Programowanie w takich językach polega na definiowaniu układów funkcji, najczęściej wzajemnie od siebie uzależnionych, a następnie wyliczaniu wartości tych funkcji dla zadanych argumentów. W czystych językach funkcyjnych, a do takich zalicza się Haskell, całkowicie nieobecne są instrukcje, w szczególności nie ma instrukcji przypisania. Nieobecność instrukcji przypisania sprawia, że zmienne w programach funkcyjnych mają taki sam charakter, jak zmienne w matematyce. Są one symbolami reprezentującymi wartości (liczbowe, logiczne, napisowe), ale gdy ustalimy, jaką wartość ma mieć dana zmienna, będzię ona mieć tę wartość przez cały okres swojej ważności. Nie można tej wartości zmienić przez wykonanie instrukcji przypisania, co jest tak charakterystyczne dla języków *imperatywnych* typu Pascal, C czy AWK. Ma to ogromne znaczenie przy konstruowaniu programów oraz myśleniu o nich.

Programy funkcyjne łatwiej niż imperatywne poddają się analizie matematycznej, gdyż obiekty w nich występujące mają prostszą interpretację matematyczną. Operuje się, jak w matematyce, na wyrażeniach algebraicznych, które w przeciwieństwie do instrukcji pozbawione są skutków ubocznych. Zwolennicy programowania funkcyjnego uważają, że właśnie skutki uboczne są poważnym źródłem błędów w programach imperatywnych. Zbliżenie zapisu programów do tradycyjnej notacji matematycznej jest główną motywacją dla rozwijania języków funkcyjnych.

Języki funkcyjne pojawiły się w świecie komputerowym już w początkach lat sześćdziesiątych, kiedy to J. McCarthy zaimplementował pierwszy z nich, o nazwie LISP. Od tamtej pory LISP został zaimplementowany na wielu maszynach. Najpopularniejsze obecnie odmiany LISP-u nazywają się SCHEME, Common LISP oraz ELISP, będący językiem skryptowym popularnego edytora GNU Emacs.

Siła współczesnych języków funkcyjnych, a zaliczyć do nich należy rozmaite odmiany języka ML oraz Haskella, bierze się z takich aspektów, jak:

- **System typów.** Każde wyrażenie ma jednoznacznie przypisany typ (który najczęściej może być wyliczony automatycznie przez komputer); każdy program przed wykonaniem jest poddany kontroli poprawności typowania występujących w nim wyrażeń;
- **Funkcje jako obiekty równoprawne z innymi.** W systemie typów funkcje są uznane za obiekty równoprawne z innymi: funkcja może wziąć inną funkcję jako swój argument, a sama może oddać funkcję w wyniku;
- **Polimorfizm.** Funkcja może posiadać typ, będący w istocie rodziną typów; w takim przypadku może ona być użyta z argumentami leżącymi w dowolnym z typów rodziny.

Matematyczne podstawy dla definicji oraz implementacji języków funkcyjnych przynosi teoria λ-*rachunku*. W przypadku LISP-u jest to rachunek, w którym zmienne nie posiadają typów (ściślej mówiąc w LISP-ie wyrażenia nie mają typów sprawdzanych statycznie, ale występuje tak zwane typowanie dynamiczne, polegające na tym, że typ wyrażenia jest z grubsza sprawdzany na etapie jego obliczania). W przypadku ML-a oraz Haskella mamy polimorficzny system typów Hindley'a–Milnera (nazwa od nazwisk autorów algorytmu pozwalającego na automatyczne wyznaczanie typów wyrażeń).

Przykład programu

Powyższy wstęp odwołuje się do wielu pojęć, z którymi nawet doświadczeni komputerowcy mogli się do tej pory nie zetknąć. Postaram się wiele z nich objaśnić na przykładach, które będą tworzyły grunt dla końcowej części artykułu, gdzie przedstawiam, jak można w Haskellu generować dane TEX-owe.

Zaczniemy od przykładu programu, w którym są zdefiniowane dwie funkcje sqr oraz dist. Pierwsza otrzymuje liczbę zmiennoprzecinkową (typu Float) jako argument i oblicza jej kwadrat. Druga operuje na punktach na płaszczyźnie, dla których oznaczania wprowadzamy pomocniczy typ Point, definiowany jako iloczyn kartezjański typów Float oraz Float. Funkcja dist oblicza odległość punktu o współrzędnych (a, b) od punktu (c, d) według znanego wzoru $\sqrt{(c-a)^2+(d-b)^2}$.

```
> module Distance where

Funkcja |sqr| oblicza kwadrat danej
liczby zmiennoprzecinkowej.

> sqr :: Float -> Float
> sqr x = x * x

Typ punktów na płaszczyźnie.

> type Point = (Float, Float)

Funkcja |dist| wylicza odległość
dwóch punktów |(a,b)| i~|(c,d)| ze
wzoru $\sqrt{(c-a)^2 + (d-b)^2}$.
Wykorzystano powyższą |sqr| oraz
standardową |sqrt|.

> dist :: Point -> Point -> Float
> dist (a,b) (c,d)
>   = sqrt (sqr (c-a) + sqr (d-b))
```

Ten program zapisaliśmy w stylu *literate*; wiersze zaczynające się od znaku '>' są kodem, a pozostałe stanowią komentarz. Programy można zapisywać także bez znaków '>'; wówczas trzeba wyróżniać komentarze, na przykład nawiasami '{-' oraz '-}'.

Powyższy program, zapisany powiedzmy w pliku Distance.lhs (lhs, bo *literate Haskell*), możemy wykonać, używając popularnego interpretera Haskella o nazwie Hugs. Wydajemy w tym celu polecenie:

```
Hugs Distance.lhs
```

Uruchomiony zostaje Hugs, który też wczytuje nasz program. Znajdujemy się teraz w trybie interpretacyjnym, z sygnałem zachęty Distance>. Możemy wyliczać wprowadzone funkcje, a przykładowy dialog mógłby wyglądać tak:

```
Distance> sqr 19
361.0
Distance> dist (1,0) (0,1)
1.41421
```

Hugs przypomina w działaniu programowalny kalkulator. Możemy wyliczać potrzebne nam wartości wyrażeń, zbudowanych z użyciem predefiniowanych w języku oraz zdefiniowanych przez nas samych operacji. Właściwie jedyną różnicą dzielącą Hugsa od elektronicznych kalkulatorów programowalnych jest bogactwo środków językowych pozostających w naszej dyspozycji.

Kilka aspektów powyższego programu wymaga komentarza. Wcześniej pisałem, że typy wyrażeń są wyliczane automatycznie, a w powyższym programie podałem typowania dla funkcji (wiersze z operatorem ::). W istocie typy dla powyższych funkcji mogłyby zostać wyznaczone automatycznie i programista nie ma obowiązku ich podawania. Prawdę powiedziawszy dla funkcji sqr komputer wyznaczyłby typ ogólniejszy niż Float, będąc niejako sprytniejszym ode mnie, bo pozwalając stosować sqr do szerszego zakresu argumentów. Byłby to typ Num a => a -> a; precyzyjne wyjaśnienie znaczenia tego napisu wykracza poza ramy artykułu, ale z grubsza wyraża on, że funkcję sqr można stosować do danych dowolnego typu a, w którym są zdefiniowane klasyczne operacje na liczbach (należących do *klasy* Num), na przykład mnożenie, dodawanie, liczba przeciwna itp.

Dlaczego typem dla dist jest Point -> Point -> Float i co to oznacza? Ten typ wyraża przede wszystkim, że dist jest funkcją dwóch argumentów, z których każdy jest punktem. Moglibyśmy też zdefiniować odległość inaczej:

```
dist' :: (Point, Point) -> Float
dist' ((a,b), (c,d))
   = sqrt (sqr (c-a) + sqr (d-b))
```

W tym przypadku funkcja byłaby jednoargumentowa, a jedynym argumentem byłaby para punktów. Rozróżnienie jest dość subtelne; być może do komputerowców przemówi taka intuicja, że chcąc obliczyć odległość dwóch punktów za pomocą funkcji `dist` przekazujemy do niej dwa punkty, jak są, podczas gdy w przypadku `dist'` trzeba najpierw w pamięci komputera z owych punktów zbudować parę.

Typ `Point -> Point -> Float` jest specyficzny pod jeszcze innym względem. Właściwie jego zapis jest skrótem od `Point -> (Point -> Float)`, gdzie bardziej widocznym się staje, że `dist` obliczona na jednym punkcie jest funkcją typu `Point -> Float`. Na przykład

```
dist0 = dist (0,0)
```

jest definicją jednoargumentowej funkcji `dist0`, obliczającej odległość punktu od początku układu współrzędnych. Możemy teraz wywołać przykładowo `dist0 (0,1)`, równorzędnie z `dist (0,0) (0,1)`. Ten przykład ilustruje jeden z aspektów charakteryzujących języki funkcyjne: funkcja może oddać funkcję w wyniku.

Typy danych w Haskellu

Siła współczesnych języków funkcyjnych, jak Standard ML czy Haskell, tkwi w ich systemie typów. W Haskellu jako podstawowe występują typ (komputerowych) liczb całkowitych `Int`, typ znaków `Char`, typ (komputerowych) liczb zmiennoprzecinkowych `Float`. Są także typy `Integer` liczb całkowitych o dowolnej precyzji oraz `Double` liczb zmiennoprzecinkowych o podwójnej precyzji. Standardowe biblioteki udostępniają działania na liczbach wymiernych `Rational` oraz zespolonych `Complex`.

Z już istniejących typów można budować bardziej złożone, wykorzystując *konstruktory typów*. Poznaliśmy już dwa konstruktory wbudowane w Haskella: konstruktor iloczynów kartezjańskich, oznaczany nawiasami i przecinkami `( , ,..., )` oraz przestrzeni funkcyjnych, oznaczany strzałką `->`. Programista może wprowadzić własne konstruktory typów używając deklaracji `type` dla skrótów (jak w przypadku `Point` powyżej) albo deklaracji `data` dla *algebraicznych typów danych*. Przykładem tych ostatnich może być deklaracja typu wartości logicznych:

```
data Bool = False | True
```

wprowadzająca typ (ściślej: zeroargumentowy konstruktor typu) `Bool` oraz dwa *konstruktory elementów* tego typu, nazwane `False` oraz `True`, będące w istocie zeroargumentowymi funkcjami o wartościach w typie `Bool` (w matematyce utarło się funkcje zeroargumentowe nazywać stałymi, ale w istocie każda funkcja wprowadzona do programu funkcyjnego jest w nim pewną stałą). Na tak zdefiniowanym typie `Bool` możemy zdefiniować na przykład negację, koniunkcję oraz alternatywę:

```
not :: Bool -> Bool
not True  = False
not False = True

(&&), (||) :: Bool -> Bool -> Bool
False && b = False
True  && b = b

True  || b = True
False || b = b
```

Krótka uwaga o notacji infiksowej oraz prefiksowej w Haskellu. Funkcji, których nazwy są zapisane za pomocą liter oraz cyfr, używa się prefiksowo, czyli pisząc nazwę funkcji przed argumentami (na przykład `not` powyżej). Jeśli funkcja, jak `&&`, ma nazwę składającą się z nieliter i niecyfr, to powinna być używana infiksowo, czyli pomiędzy argumentami. W powyższej definicji użyliśmy notacji infiksowej nawet po lewej stronie równań definiujących koniunkcję. Można te konwencje obejść. I tak, `dist` może być użyta infiksowo, jeśli nazwę zapiszemy z apostrofami `` `dist` ``. Z kolei `(&&)` jest rozumiane jako operator prefiksowy.

Powyżej zdefiniowane koniunkcja oraz alternatywa unikają wyliczania drugiego argumentu, gdy wartość pierwszego wystarcza do wyznaczenia wyniku. Na przykład, `False && b` ma wartość `False` także gdy obliczenie `b` wywołuje nieskończoną pętlę. Ta cecha obliczeń w Haskellu nosi nazwę *leniwości*: argument funkcji nie jest wyliczany w chwili jej wywołania, a dopiero wtedy, gdy jego wartość jest niezbędna do wyznaczenia wyniku wywołania. Analiza rozmaitych konsekwencji leniwości – pożytecznych jak powyżej, ale niekiedy szkodliwych – wykracza poza ramy tego artykułu.

Listy

Ciągi elementów występują jako podstawowa struktura danych praktycznie we wszystkich językach programowania. W językach imperatywnych ciągi pamiętane są w tablicach, ale jest też możliwość budowania list „dynamicznie”, posługując się wskaźnikami, *de facto* adresami elementów. W językach funkcyjnych sytuacja jest, rzec by można, dualna. To listy dynamiczne wysuwają się na plan pierwszy, zaś tablice są traktowane po macoszemu. Z tym, że zarządzanie owymi wskaźnikami oraz cała gospodarka pamięcią podczas wykonania programu spoczywa na interpreterze – nie musimy myśleć o rezerwowaniu ani zwalnianiu pamięci.

Zapiszmy kilka przykładów ciągów w postaci list Haskellowych:

```
[]
1 : (0 : (-3 : []))
True : False : True : []
dist (0,1) : dist (0,0) : []
'A' : 'l' : 'a' : []
(5 : 2 : []) : (7 : []) : [[]] : []
```

Na początku zapisaliśmy ciąg pusty, oznaczony parą nawiasów `[]`. Każda lista w programie Haskellowym jest bądź pusta, bądź powstaje przez dopisanie nowego elementu do listy już istniejącej. W drugim przykładzie do listy pustej najpierw dopisano `-3`, następnie do jednoelementowej listy `-3 : []` dopisano 0, a na koniec jeszcze dopisano 1, uzyskując listę trójelementową. W trzecim przykładzie na liście są trzy wartości logiczne, kolejno `True`, `False` oraz `True` (opuściliśmy nawiasy, zgodnie z konwencją, że operator dołączania `:` wiąże silniej z prawej strony). W czwartym na liście są dwie funkcje, w piątym są znaki, a w szóstym mamy listę list.

Chociaż nie ma innego sposobu budowania list niż przez użycie zeroargumentowego operatora `[]` lub dwuargumentowego operatora `:`, to istnieje szereg konwencji notacyjnych do wygodniejszego zapisywania wyrażeń listowych. Przez konwencję notacyjną należy rozumieć, że podczas interpretacji programu następuje automatyczna konwersja formy skróconej do formy podstawowej. Oto równoważne formy skrótowe dla wybranych przykładów:

```
[1,0,-3]
['A','l','a']
"Ala"
[[5,2], [7], []]
```

Istnieją jeszcze inne ułatwienia notacyjne. Na przykład `[1..33]` oznacza wszystkie liczby całkowite od 1 do 33. Lista `['a'..'z']` zawiera wszystkie małe litery od `'a'` do `'z'`. Zapis

```
[sqr n | n <- [1..25], n `mod` 3 = 0]
```

oznacza listę kwadratów wszystkich liczb od 1 do 25 (pobieranych z listy `[1..25]`), podzielnych przez 3.

Typ list. Operatory listotwórcze `[]` oraz `:` mogą być stosowane do elementów różnych typów. Tak były używane w przykładach powyżej. Nasuwa się pytanie, czy operatorom tym oraz listom za ich pomocą budowanym można jednoznacznie przypisać typ.

W Haskellu typ list o elementach typu `a` jest oznaczany przez `[a]`. Tak więc lista `['A','l','a']` ma typ `[Char]`, typem dla `True:False:[]` jest `[Bool]`, zaś typ `[dist (0,1)]` to `[Point -> Float]`.

Mimochodem użyłem litery `a` na oznaczenie typu elementów listy, a następnie podstawiałem pod ten symbol różne znaczenia. Raz `a` było rozumiane jako `Char`, innym razem jako `Point -> Float`. Symbol `a` zagrał rolę *zmiennej typowej*, o wartościach w zbiorze typów.

Typ list jest w Haskellu zdefiniowany za pomocą równania rekurencyjnego

```
data [a] = [] | a : [a]
```

Natura tego równania wyraża to, co wcześniej napisaliśmy o budowaniu list. Lista elementów typu `a` jest albo pusta, albo powstaje przez zastosowanie operatora `:` do wartości typu `a` oraz (rekursja na typach) listy typu `[a]`. Nie wypowiadamy się, czy listy przez nas budowane będą składać się liczb, par, znaków itd. Równanie powyższe stosuje się do dowolnego typu reprezentowanego przez `a`. Zmienna `a` jest w równaniu jak gdyby objęta *kwantyfikatorem ogólnym* podobnie, jak x w formule logicznej $\forall x(x = x)$.

O typach, których definicje korzystają ze zmiennych typowych, mówimy że są *polimorficzne*. Można myśleć, że typ polimorficzny to rodzina albo schemat typów, chociaż w Haskellu mówimy o jednym typie. Operatory `[]` oraz `:` są *funkcjami polimorficznymi*; ich typowanie przebiega z użyciem zmiennej typowej:

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

Drugą klauzulę można odczytać następująco. Operator `:` jest funkcją, która biorąc element pewnego typu – oznaczmy go literą `a` – oraz listę elementów typu `a` oddaje w wyniku listę elementów typu `a`. Naturalnie samo wyrażenie typujące nie oddaje, jaka lista powstaje w wyniku zastosowania operatora; z Haskellowej definicji operatora : wiemy, że będzie to lista z dołączonym na początku nowym elementem. Typowanie jest jednak pomocne przy analizie poprawności wywołań funkcji.

Funkcje `head`, `tail`, `gentleTail`. Bardzo często przydaje się, dostępna jako funkcja, operacja pobrania pierwszego elementu z listy:

```
head :: [a] -> a
head (x:xs) = x
```

Dopełnieniem do `head` jest `tail`, wydobywająca z listy niepustej jej „ogon", czyli listę bez pierwszego jej elementu:

```
tail :: [a] -> [a]
tail (x:xs) = xs
```

Definicje funkcji head oraz `tail` zawierają wyłacznie klauzule dla list niepustych. Oznacza to, że próba zastosowania którejkolwiek z tych funkcji do listy pustej zakończyłaby się błędem, sygnalizowanym podczas wykonania programu. W przypadku `tail` jest możliwa wersja łagodniejsza, dająca wynik także dla listy pustej, ale też ze zmienionym znaczeniem pojęcia „ogon":

```
gentleTail :: [a] -> [a]
gentleTail []     = []
gentleTail (x:xs) = xs
```

Powyższe funkcje są polimorficzne, czyli możemy je stosować do argumentów różnych typów. Przykłady: `head [1]` da w wyniku `1`, a wynikiem `tail " Las"` będzie napis "Las".

Rekursja, wyższe rzędy

Rekursja nie powinna być komputerowcom obca, ale warto powiedzieć, że nieobecność w języku jakichkolwiek instrukcji oznacza między innymi brak iteracji. Tam, gdzie w języku imperatywnym do zapętlenia obliczenia użylibyśmy pętli `while...`, `repeat...`, `for...`, czy jak się to w nim pisze, w języku funkcyjnym musimy użyć rekursji.

Przez funkcje wyższych rzędów rozumie się takie, które otrzymują inne funkcje jako argumenty. Możliwość przekazywania funkcji jako argumentów funkcji jest ważnym środkiem modularyzacji w programowaniu fukcyjnym; to jest wręcz istota programowania funkcyjnego!

Funkcja `map`. Przypuśćmy, że chcemy pomnożyć każdy element listy przez 2. Możemy w tym celu posłużyć się funkcją `map`, której definicję za chwilę poznamy. Na przykład

```
map (* 2) [1,-3,4]
```

da nam w wyniku `[2,-6,4]` (w tym miejscu ponownie skorzystaliśmy z Haskellowej możliwości uzyskania funkcji jednoargumentowej przez zastosowanie funkcji dwuargumentowej do ustalonego argumentu `(* 2)`).

Zanegować elementy listy o wartościach logicznych można tak:

```
map not [True,False]
```

W wyniku uzyskamy `[False,True]`. Oto, jak można obliczyć odległości poszczególnych punktów w danym ciągu od punktu (0,0):

```
map dist0 [(0,1), (1,1), (1,0)]
```

W każdym z powyższych przykładów funkcja `map` była stosowana do argumentów innego typu. Domyślamy się, że jej definicja powinna być polimorficzna. Na dodatek jako pierwszy argument występowała funkcja, tak więc `map` jest funkcją wyższego rzędu. Oto definicja:

```
map :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Schemat rekursji w tej definicji przebiega po strukturze listy. Dla listy pustej `[]` nie ma elementów, do których należałoby stosować `f`, więc wynik jest pusty. Dla listy niepustej, czyli takiej, którą z racji ogólnej zasady konstruowania list da się rozłożyć na pierwszy element `x` oraz resztę listy `xs`, budujemy wynik przez zastosowanie funkcji `f` do `x` i doczepienie uzyskanej wartości do listy `xs` przekształconej tą samą metodą (rekursywne wywołanie `map f xs`).

Składanie funkcji. Odpowiednikiem następstwa instrukcji, najczęściej oznaczanego w językach imperatywnych średnikiem, jest w językach funkcyjnych złożenie funkcji. Zapewne pamiętacie ze szkoły lub ze studiów, jak się składa funkcje. Najpierw działamy pierwszą funkcją, a do wyniku stosujemy drugą. Oczywiście musimy wiedzieć, że ta druga może działać na wynikach pierwszej. W Haskellu złożenie dwóch funkcji oznacza się kropką:

```
(.) :: (b->c) -> (a->b) -> (a->c)
(g . f) x = g (f x)
```

Tutaj najpierw jest stosowana `f`, dająca wyniki w typie `b`, a do jej wyniku aplikujemy `g`, mającą argumenty w tymże typie `b`. Przykładem użycia może być złożenie `sqr . dist0`. Jej typem jest `Point -> Float`, a oblicza ona kwadrat odległości danego punktu od początku układu współrzędnych.

Zwijanie list. Przyjrzyjmy się następującym trzem definicjom:

```
sum :: [Int] -> Int
sum []         = 0
sum (x:xs)     = x + sum xs

product :: [Int] -> Int
product []     = 1
product (x:xs) = x * product xs

and :: [Bool] -> Bool
and []         = True
and (x:xs)     = x && and xs
```

Schemat działania jest w każdym z przypadków podobny. Gdy lista jest pusta, wynikiem jest odpowiednia stała typu wynikowego (w podanych funkcjach jest to element neutralny wykonywanego działania dwuargumentowego). Gdy lista jest niepusta, stosujemy dwaurgumentowy operator do pierwszego jej elementu oraz do wyniku rekurencyjnego obliczenia funkcji na reszcie listy. Możemy zdefiniować polimorficzną funkcję wyższego rzędu, uogólniającą powyższe schematy:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Nazwa `foldr` jest skrótem od angielskiego *fold from the right*, czyli zwijaj od prawej. Teraz możemy zdefiniować powyższe funkcje w krótszej formie:

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
```

Możemy wykorzystać `foldr` do sprawnego definiowania innych operacji, na przykład do wyliczania długości listy, czyli liczby jej elementów:

```
length :: [a] -> Int
length = foldr add1 0
   where add1 x k = k + 1
```

Pomocnicza funkcja `add1` oddaje w wyniku wartość drugiego argumentu powiększoną o 1. Wartość pierwszego argumentu jest ignorowana. W ten sposób każdemu wystąpieniu konstruktora `:` w liście odpowiada dodanie jedności. Poprawność powyższej definicji, czyli że `length` rzeczywiście wyznacza długość listy, powinna być zatem oczywista.

Definicja funkcji `length` ilustruje wykorzystanie pewnego aspektu typowania funkcji `foldr`. Uważny Czytelnik może wyrazić zdziwienie, dlaczego w typie `foldr` występują dwie zmienne typowe, `a` oraz `b`. W przypadku `sum`, `product` oraz `and` zwijaliśmy listy, stosując odpowiednio operacje `+`, `*` oraz `&&`, z których każda działa na argumentach tego samego typu. Mogłoby się wydawać, że typem `foldr` powinno być wyrażenie:

```
(a -> a -> a) -> a -> [a] -> a
```

Ale schemat, według jakiego działa `foldr` nie narzuca takiego ograniczenia, pozwalając na uogólnienie: pierwszy i drugi argument stosowanego operatora mogą być różnych typów. Już to wykorzystaliśmy, definiując `length`. Do znajdywania możliwie najogólniejszego typowania interpretery i kompilatory używają algorytmu Hidley'a–Milnera. Tenże algorytm wyliczyłby typowanie ze zmiennymi `a` i `b`. Co sprytniejsi Czytelnicy powinni być w stanie na własną rękę wyznaczyć ów typ, przez sformułowanie minimalnych ograniczeń na zmienne typowe w wyrażeniu `a -> b -> c -> d` (funkcja `foldr` ma trzy argumenty, więc od takiego wyrażenia można zacząć), aby uzgodnić typ z postacią równań.

Jako wymagające chwili namysłu ćwiczenie dla wszystkich proponuję rozstrzygnięcie, co wyliczają poniższe funkcje. Dla ułatwienia podaję odpowiednie typowania:

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) xs ys

concat :: [[a]] -> [a]
concat = foldr (++)
```

Grupowanie elementów w liście. Przy operowaniu na listach często zachodzi potrzeba klasyfikowania elementów w nich zawartych ze względu na zadane właściwości. W dalszym ciągu przyda nam się kilka takich funkcji. Funkcja `span` przegląda listę w poszukiwaniu pierwszego, który nie spełnia zadanego warunku. Wynikiem są dwie listy; na pierwszej z nich wszystkie elementy spełniają warunek, pierwszy element drugiej (jeśli taki istnieje) nie spełnia warunku. Funkcja `break` wykonuje podobne zadanie, ale przystępuje do przeglądania z zamiarem znalezienia pierwszego elementu, który spełnia warunek.

```
span, break :: (a -> Bool) -> [a] ->
                              ([a],[a])
span p [] = ([],[])
span p (x:xs) | p x = (x:xs', xs'')
        where (xs',xs'') = span p xs
span p xs = ([],xs)

break p =  span (not . p)
```

Kilka uwag o tych definicjach. W przypadku listy pustej nie ma ani elementów spełniających predykat p, ani spełniających jego zaprzeczenie. Gdy lista jest niepusta, a jej pierwszy element spełnia predykat p (notacja z kreską | w drugiej klauzuli oznacza sprawdzenie tego warunku), to ten element rozpoczyna listę elementów spełniających p. Należy go doczepić na początek pierwszej składowej pary wynikającej z działania span na reszcie elementów. Trzecia klauzula w definicji span odnosi się do przypadku listy niepustej, w którym pierwszy element nie spełnia predykatu. Istotnie, po rozpatrzeniu pierwszej klauzuli w kolejnych mamy do czynienia z listą niepustą, a po rozpatrzeniu drugiej wiemy, że pierwszy element listy niepustej nie spełnia predykatu. Tak więc w ostatnim przypadku pierwsza składowa wyniku powinna być pusta, a druga zawierać całą listę.

W definicji funkcji break wykorzystaliśmy wprowadzoną wcześniej operację składania funkcji. Naturalnie w miarę rozrastania się naszej biblioteki, coraz więcej pojawia się sytuacji, gdy możemy wykorzystać wcześniej zbudowane funkcje. Obecność w języku funkcji wyższych rzędów oraz polimorfizmu ułatwia ponowne wykorzystanie mozolnie wypracowanych algorytmów.

Oto jeszcze jedna funkcja klasyfikująca elementy:

```
groupBy :: (a -> a -> Bool) -> [a] ->
                                  [[a]]
groupBy eq [] =  []
groupBy eq (x:xs)
  = (x:ys) : groupBy eq zs
     where (ys,zs) = span (eq x) xs
```

Tym razem chodzi o podział elementów listy na grupy (wynikiem jest lista grup, reprezentowana jako lista list). W pierwszej grupie kolejne, sąsiadujące ze sobą, elementy spełniają zadaną relacją dwuargumentową. Gdy napotkamy element, który nie jest w relacji z poprzednim, tworzy się nowa grupa. Na przykład, gdy relacją jest porównanie ==, to w każdej grupie znajdą się elementy sobie równe. Gdy relacją będzie porównanie <, każda grupa będzie ciągiem rosnącym, rozciągającym się w oryginalnej liście do pierwszego elementu zaburzającego wzrost.

Zastosowania z TEX-em w tle

Jak zapewne Czytelnicy się zorientowali, Haskell jest językiem *ogólnego zastosowania*. Nie jest on ukierunkowany na rozwiązywanie szczególnej kategorii zagadnień, jak to jest w przypadku AWKa, PERLa czy samego TEX-a. Wydaje się, że na styku z TEX-em może okazać się przydatny wszędzie tam, gdzie rozwiązanie danego zagadnienia w języku specjalizowanym jest kłopotliwe, na przykład wymaga stosowania pokrętnych trików, obcych naturze języka. Autor niniejszego artykułu wykorzystuje Haskella wszędzie tam, gdzie dane tekstowe występują łącznie z innymi rodzajami danych: liczbowymi, logicznymi, algebraicznymi itp, także tam, gdzie z danych tekstowych trzeba wydobyć i przetworzyć informację o innym charakterze, jako fazę przejściową przed generowaniem danych dla TEX-a.

Konkretnymi przykładami mogą być:

- program mający na wejściu tekst przeznaczony dla TEX-a, w którym przed TEX-ową kompilacją trzeba wyliczyć wartości pewnych wyrażeń arytmetycznych (rachunki, faktury);
- terminarzyki;
- sortowanie odpowiedzi w zbiorze zadań;
- rozwiązywanie łamigłówek typu „obrazki logiczne" z ich prezentacją za pomocą pakietu tap panów B.L. Jackowskiego, P. Pianowskiego i P. Strzelczyka;
- przygotowanie tekstów *literate Haskell* do składu w LATEX-u.

Niektóre z tych programów oraz kilka innych można znaleźć pod adresem ftp:

`ftp.ipipan.gda.pl/rysiek/HugsProgs`

Omówię teraz strukturę programu rozwiązującego ostatnie z wymienionych zagadnień. Dla celów prezentacyjnych przedstawiona poniżej wersja programu, jak i samo zagadnienie, zostały okrojone. Pełny program jest dostępny pod powyższym adresem.

Przykładem tekstu w stylu *literate Haskell* jest podany wcześniej moduł Distance. Jak wspominałem, wiersze kodu zaczynają się w nim od znaku

'>'. Te wiersze chcę drukować *verbatim*, z zachowaniem oryginalnego spacjowania w programie. Miłośnicy stylu *literate* mogą czuć się zawiedzeni, bo skład uzyskany z 'weave' (jedna z podstawowych składowych 'web2c' – pakietu programów do obróbki tekstów *literate* C) ma nazwy zmiennych kursywą, słowa kluczowe są w nim wytłuszczone i w ogóle jest bardzo elegancko. Mnie wystarcza, że elegancko złożona zostaje część komentarzowa, czyli wiersze nie zaczynające się od znaku >, dodatkowo oddzielane od kodu pustymi wierszami. Ponieważ na etapie tworzenia jakiegokolwiek programu nieustannie pracuje się nad kodem od strony merytorycznej, konieczność dodawania wskazówek odnośnie składu rozprasza. Nad eleganckim składem całości warto popracować na potrzeby wersji finalnej.

W wierszach niekodowych mogą występować wstawki, które podobnie jak kod należy złożyć pismem maszynowym. Wstawki te są ograniczone kreskami |, jak |sqr| w module Distance. W przypadku, gdy sam znak kreski występuje wewnątrz tekstu, można użyć większej liczby kresek na początku i równej im liczby kresek na końcu.

Zadanie przed nami stojące jest więc takie. Wejściowy tekst dociera w postaci ciągu znaków, czyli [Char], czyli String. Ten ciąg dzielimy na kolejne wiersze, uzyskując listę typu [String]. Wiersze te należy poklasyfikować pod kątem ich przynależności do tak zwanych „kęsów" (*chunk*). Wprowadźmy dla zwięzłości oznaczenie

```
type Chunk = [String]
```

Mamy kęsy kodowe, niekodowe oraz takie, w których każdy wiersz zawiera tylko znaki niewidoczne. W wyniku klasyfikacji będziemy mieli do dyspozycji listę typu [Chunk]. Następnie każdy z kęsów trzeba przekształcić, dodając w nim odpowiednie polecenia LaTeX-owe. Całość trzeba jeszcze uzupełnić o właściwy dla LaTeX-a początek i koniec dokumentu. Na sam koniec należy wyniki skonwertować do postaci String.

Główna funkcja programu, realizująca kolejno powyższe kroki, może zostać przedstawiona w postaci złożenia funkcji podrzędnych:

```
lhs2ltx :: String -> String
lhs2ltx = concat
        . map unlines
        . ([preamble] ++)
        . (++ [postamble])
        . map transformChunk
        . splitToChunks
        . lines
```

Pamiętamy, że składane funkcje są wykonywane od ostatniej w ciągu do pierwszej. Tak więc najpierw tekst jest dzielony na wiersze za pomocą lines, potem następuje podział na kęsy oraz ich przekształcenie, dodane zostają początek i końcówka pliku LaTeX-owego. Na koniec następuje powrót do postaci ciągu znaków, oczywiście z nową już zawartością.

Poniżej podaję definicje wybranych z brakujących elementów układanki. Pominięte są między innymi kwestie wywołania programu z ciągiem znaków zawartym w pliku dyskowym. Szczegóły są dostępne pod wcześniej podanym adresem elektronicznym.

Funkcje `lines` **oraz** `unlines`. Pierwsza z nich wyszukuje w tekście znaki zmiany wiersza '\n'. Każdy taki znak oznacza zakończenie rozpoznawania wiersza. Wiersz zostaje dołączony do wyniku podziału reszty, a na pozostałej części tekstu (po usunięciu znaku zmiany wiersza) procedura się powtarza. Druga z funkcji jest odwrotna do pierwszej, z tym że na końcu na pewno znajdzie się znak zmiany wiersza, którego przed podziałem mogło w tekście nie być.

```
lines :: String -> [String]
lines "" = []
lines xs = l : lines (gentleTail ys)
  where
    (l,ys) = break (== '\n') xs

unlines :: [String] -> String
unlines = concat . map (++ "\n")
```

Stałe `preamble` **oraz** `postamble`. Na mocy decyzji projektowej obydwie stałe mają być ciągami napisów, z których każdy będzie zamieniony na jeden wiersz w wynikowym dokumencie. Trzeba odpowiednio rozpocząć oraz zakończyć dokument LaTeX-owy, a do prembuły powinny trafić ustawienia parametrów LaTeX-a. Przykładowo może to wyglądać następująco:

```
preamble, postamble :: [String]
preamble =
 [ "\documentclass{article}"
 , "\usepackage{a4wide}"
 , "\usepackage{polski}"
```

```
  , "\begin{document}"
  ]

postamble = [ "\end{document}" ]
```

Funkcja splitToChunks. Po podzieleniu tekstu wejściowego na wiersze, grupujemy je za pomocą groupBy w taki sposób, że do jednej grupy trafią kolejne wiersze

- składające się tylko ze znaków niewidocznych;
- zaczynające się od znaku '>';
- zaczynające się od innego znaku niż '>'.

Być może poniższa definicja prowadzi niepotrzebnie do powtórnego sprawdzania niektórych warunków; Czytelnik może pomyśleć o sprawniejszej realizacji.

```
splitToChunks :: [String] -> [Chunk]
splitToChunks = groupBy inOneChunk

inOneChunk :: String -> String -> Bool
inOneChunk s t
  = all isSpace s
    && all isSpace t
  || not (all isSpace s)
    && head s == '>'
    && not (all isSpace t)
    && head t == '>'
  || not (all isSpace s)
    && head s /= '>'
    && not (all isSpace t)
    && head t /= '>'
```

Użyty tu predykat isSpace :: Char -> Bool sprawdza, czy dany znak jest odstępem, tabulacją lub innym znakiem niewidocznym.

Funkcja transformChunk. To właśnie ona dokonuje istotniejszych zmian w tekście, dodając doń polecenia sterujące składem. Każdy z rodzajów kęsów jest traktowany swoiście, czyli raz jeszcze nastąpić powinno rozpoznanie rodzaju kęsu (tym razem wystarczą prostsze testy) i zależnie od wyniku zastosowanie odpowiedniego przekształcenia.

```
transformChunk :: Chunk -> [String]
transformChunk c
  | c == [] || all isSpace (head c) = c
  | head (head c == '>') = transformCode
  | otherwise = transformComment
```

Jak widać, wiersze niewidoczne przepuszczamy bez zmiany, a pozostałe przekształcamy zależnie od rodzaju. W przypadku kodu, intencją jest, by składać go w LaTeX-owym środowisku verbatim.

```
transformCode :: Chunk -> Chunk
transformCode xss
   =  ["\\begin{verbatim}"]
   ++ map line xss
   ++ ["\\end{verbatim}"]
    where
     line xs = "    " ++ tail xs
```

Z każdego wiersza mają znikają początkowe znaki '>', a zamiast nich jest generowane zgrabne wcięcie; te zadania wypełnie pomocnicza lokalna funkcja line :: String -> String.

Definicję funkcji transformComment pomijam. Przy jej programowaniu należy uwzględnić, że komentarze powinny być składane jak zwykły tekst LaTeX-owy, ale trzeba w nich wyłowić wstawki kodowe, jak |dist|. Każda taka wstawka powinna być wypuszczona do składu w wersji verbatim.

Skąd zdobyć materiały o Haskellu

Główna Internetowa strona Haskella ma adres:

http://www.haskell.org

Jedną z grup dyskusyjnych związanych z programowaniem fukcyjnym jest

comp.lang.functional

Członkowie tej grupy prowadzą listę często zadawanych pytań i odpowiedzi:

http://www.cs.nott.ac.uk/Department/Staff/gmh/faq.html

Prosty w użyciu interpreter Haskella o nazwie Hugs, w wersjach dla systemów wyrosłych z DOSa oraz UNIXa, można ściągnąć spod adresu ftp:

ftp.cs.nott.ac.uk/pub/haskell/hugs

Powstałe do tej pory kompilatory Haskella ('ghc' z Glasgow, 'hbc' z Göteborga – odsyłacze do nich są na głównej stronie Haskella) są w zasadzie przeznaczone dla systemów UNIXowych.

Podziękowania. Dziękuję Kolegom Włodkowi Bzylowi i Piotrowi Strzelczykowi za wnikliwe zapoznanie się z jedną z wersji artykułu i przekazanie mi cennych uwag.

Prace nad zastosowaniem Haskella do generowania danych TeX-owych były częściowo finansowane przez pakiet LoSSeD w ramach projektu CRIT-2, fundowanego przez programy ESPRIT oraz INCO.

◇ Ryszard Kubiak
r.kubiak@ipipan.gda.pl