Zadanie 1 - W tym zadaniu będzie zapewne mniej pisania, niż w poprzednich "zwykłych", ale za to trzeba będzie się dużo więcej zastanawiać, co tu się właściwie dzieje. Wyjątkowo możesz nie przejmować się wyciekami pamięci – ale nadal należy sprawdzać, czy dereferencjonowane wskaźniki nie są NULL (albo typu void *).

Do zadania dołączony jest plik nagłówkowy oraz program testowy korzystający ze zdefiniowanych w nim funkcji. Należy napisać implementację tych funkcji tak, by możliwa była poprawna rozdzielna kompilacja, konsolidacja, i wykonanie programu.

Funkcje mają służyć do obsługi list wiązanych zawierających elementy (teoretycznie) dowolnego typu, gdzie polimorfizm uzyskujemy za pomocą wskaźników (prowadzących – najczęściej – na stertę, nawet jeśli jest to wskaźnik do pojedynczej liczby), dla których zapamiętujemy (za pomocą wartości typu char) typ przechowywanej tam wartości. Służy do tego zdefiniowana w pliku nagłówkowym krotka PWT. W tym zadaniu będziemy umieszczać na listach tylko liczby (typu long, znacznik typu 'i') i napisy (typu char *, znacznik typu 's').

Listy składające się z krotek typu struct listelem będą tworzone za pomocą funkcji cons. W nowym elemencie listy umieszczamy wartość typu PWT i nie musimy tworzyć nowej kopii danych znajdujących się w referencjonowanym miejscu. Powinniśmy za to sprawdzać, czy lista zawiera elementy tylko jednego typu (chcemy mieć listy napisów i listy liczb, ale nie listy mieszane). Lista pusta jest tu (naturalnym) wyjątkiem – jej "logicznego" typu nie da się określić. cons może się nie powieść (przez błąd przydziału pamięci) – wtedy oczywiście zwracaj NULL.

Właściwie wszystkie pozostałe funkcje będą rekurencyjne. Funkcja map jest "funkcją wyższego rzędu" – przyjmuje listę i (wskaźnik na) funkcję, i tworzy listę odpowiadającą otrzymanej, ale której wszystkie elementy zostaną "przepuszczone" przez podaną funkcję. Dzięki niej będzie można napisać funkcje takie, jak take_abs (tworzącą listę liczb zamienionych na dodatnie) czy take_strlen (tworzącą listę długości napisów z oryginalnej listy) w taki zwięzły sposób (sama map też może mieć dwie linijki, wykorzystując cons):

```
list take_abs(list l) {
   return map(l, abs_aux);
}
list take_strlen(list l) {
   return map(l, strlen_aux);
}
```

Niestety pomocnicze funkcje abs_aux i strlen_aux nie będą już aż tak eleganckie – na pewno będą wykorzystywały abs i strlen (lub ich nasze odpowiedniki), ale muszą mieć odpowiedni typ (przyjmują i zwracają PWT, a nie liczby i napisy), sprawdzać, czy "logiczny" typ argumentów (zapisany w polu type) jest odpowiedni, i przydzielać pamięć,

do której odwoływać się będzie pole ptr zwracanej wartości. W przypadku jakiegokolwiek niepowodzenia wystarczy, że funkcje te będą zwracały np. (PWT) {NULL, 'n'} (zwróć uwagę na inny znacznik typu niż wcześniej wymieniane).

Kolejne funkcje wyższego rzędu to "foldy" foldl i foldr (lewy i prawy). Przyjmują one dwuargumentowe funkcje oraz jedną początkową wartość, a potem "zwijają" wartości z listy na tę początkową wartość (używając podanej funkcji) – foldl od głowy listy, a foldr od ogona. Przy ich użyciu można (w miarę) łatwo napisać szereg kolejnych funkcji, np. len (zwracająca długość listy – tu wyjątkowo typ jej elementów nie będzie istotny), sum (zwracająca sumę elementów listy liczb), czy totlen (łączna długość napisów na liście napisów). Będą to nie tyle jednolinijkowce, co… trzylinijkowce (?), gdzie najważniejsza instrukcja będzie wyglądać jakoś tak:

PWT tmp = foldl(l, /* ??? */, len_aux);

przy czym najpierw być może trzeba będzie przygotować odpowiednią "wartość początkową" (typu PWT), a na końcu spytać, czy tmp ma odpowiedni znacznik typu i odpowiednio wyłuskać wartość do zwrócenia (zauważ, że te funkcje już mają nie zwracać PWT, co ma zresztą pewien mankament, ale już może wystarczy…).

Oczywiście znowu trzeba będzie przygotować funkcje pomocnicze len_aux, sum_aux, totlen_aux itp. które znowu m.in. będą sprawdzać, czy ich argumenty mają właściwe znaczniki typów. Zauważ, że totlen_aux ma ciekawy "logiczny" typ (zwraca liczbę, ale przyjmuje liczbę i napis), i że len_aux w zasadzie mogłaby ignorować jeden ze swoich argumentów (na co marudziłby kompilator, ale zamiast wyłączać jego marudzenie globalnie, wystarczy że dodamy w kodzie tej funkcji instrukcję (void) nazwa_nieuzywanego_argumentu; i tym mu zamydlimy oczy).

Funkcja cat ma skleić napisy z listy w jeden. Ją też chcemy zaimplementować przy użyciu któregoś z foldów, choć warto zauważyć, że bez dodatkowej ostrożności (która, jak napisano wyżej, nie jest tym razem wymagana), wszystkie "pośrednie" napisy (w naszym przykładowym programie np. "AlaMa" lub "MaKota", zależnie od wybranego kierunku zwijania) zostaną utworzone na stercie i porzucone. Trudno.

Zauważ, że dla funkcji łącznych kierunek zwijania nie ma znaczenia – ale nie wszystkie funkcje takie są. Okazuje się, że również operację odwracania listy można zrealizować za pomocą foldów, a konkretnie foldl. Funkcją pomocniczą będzie tu… cons, odpowiednio owinięty tak, by przyjmował i zwracał PWT. Tu do sprawdzania poprawności wywołania przyda się wprowadzenie kolejnego znacznika typu, np. 'l'. (Gdybyśmy użyli w podobny sposób foldr, otrzymalibyśmy… (płytką) kopię listy.)

Ostatecznie okazuje się, że foldl można w naturalny sposób zaimplementować iteracyjnie. Zrób również i to, umieszczając w kodzie oba warianty tak, by można było pomiędzy nimi wybierać – podobnie jak w jednym z poprzednich zadań – przez

kompilację warunkową i flagę kompilatora -DFOLDL_ITER. (Jeśli zamierzasz zaimplementować tylko jeden wariant, to oczywiście nie wprowadzaj kompilacji warunkowej, a ww. flaga będzie ignorowana.)

Wskazówka: zauważ, że funkcje map, foldl i foldr nie są jawnie używane w głównym programie. Możesz więc napisać funkcje take_abs, sum, rev itd. "na piechotę", jeśli ww. zabawy z przekazywaniem funkcji do funkcji są zbyt zagmatwane. Oczywiście za takie rozwiązanie nie zostaną przyznane pełne punkty.

Uwagi: to podejście wywala do kosza dużą część systemu typów języka C i opiera się na optymistycznym założeniu, że nasze funkcje same będą poprawnie sprawdzać typy w runtime, jak w jakimś Pythonie. map też da się zaimplementować za pomocą foldr, ale operowanie na funkcjach w C jest na tyle toporne (przez np. brak naturalnej aplikacji tylko części argumentów), że chyba nie da się tego zrobić elegancko (zakładając, że to, co już w tym zadaniu jest, jeszcze jest eleganckie). To i wiele innych podobnych rzeczy zobaczą Państwo w bardziej naturalnym kontekście na metodach programowania i ew. programowaniu funkcyjnym (które gorąco polecam).

Zadanie 2. Na pewnej planecie znajduje się n miast, połączonych m głównymi dwukierunkowymi trasami: lotniczymi, kolejowymi i autobusowymi. Turysta Ijon Tichy nie ma zbyt wiele czasu przed kolejną podróżą międzygwiezdną, a koniecznie chciałby przetestować po jednym rodzaju połączenia. Powziął więc zamierzenie, że najpierw uda się gdzieś z miasta nr 1 samolotem, następnie wybierze pojedynczy przejazd koleją, by zakończyć podróż autobusem. Innymi słowy, chce przemieścić się z miasta nr 1 trzema połączeniami, korzystając w odpowiedniej kolejności ze wszystkich trzech środków transportu. Zastanawia go jednak, do ilu różnych miast może w ten sposób dotrzeć (przy czym nie przeszkadza mu, gdyby miał w trakcie podróży się 'cofać'). Dodatkowo, gdyby dostał wcześniejsze wezwanie do opuszczenia planety, Ijona interesuje jeszcze druga liczba - do ilu miast można dotrzeć z miasta nr 1 korzystając tylko z (nieograniczonej liczby, wliczając zero) połączeń lotniczych.

Wejście
W pierwszym wierszu wejścia znajdują się naturalne liczby 1 < n,m < 100000. W każdym z kolejnych m wierszy wejścia znajduje się ciąg x y z, gdzie x,y to liczby będące numerami dwóch różnych miast (zatem z zakresu [1,n]), a z to pojedyncza litera 'L', 'K' lub 'A', oznaczające środek transportu łączący miast o numerach x,y. Potencjalnie między dwoma miastami może występować wiele połączeń.

Wyjście
Na standardowe wyjście należy wypisać dwie oddzielone spacją liczby, będącą liczbą miast do których może dotrzeć kosmiczny turysta na wskazane sposoby.

Przykłady
Przykład A

Wejście
2 1
1 2 L

Wyjście
0 2

Na planecie brak połączeń kolejowych czy autobusowych, zatem pierwsza metoda jest niewykonalna. Da się jednak przemieścić do miast 1 i 2 za pomocą samolotu.

Przykład B
Wejście
4 3
1 2 L
2 3 K
3 4 A

Wyjście
1 2

Trasa z miasta 1 do 4 spełnia wymogi wykorzystania kolejno samolotu, pociągu i autobusu. Ponadto nadal do miast 1 i 2 można dotrzeć samolotami.

Przykład C
Wejście
5 7
1 2 L
1 2 K
1 3 L
2 4 K
3 4 K
1 5 A
2 4 A

Wyjście
2 3

Możemy dotrzeć do miast 2 i 5, wymieniając środki transportu. Miasta 1, 2 i 3 są dostępne przy użyciu jedynie samolotów.