Indywidualny projekt programistyczny (Info, I rok) 19/20

Małe zadanie

Na świecie rośnie wiele lasów. W każdym lesie jest wiele drzew. Na każdym drzewie może żyć wiele zwierząt. Każdy las ma unikalną nazwę. Każde drzewo w danym lesie ma unikalną nazwę. Każde zwierzę żyjące na danym drzewie ma unikalną nazwę. Napisz program odwzorowujący ten świat.

Opis działania programu

Po uruchomieniu programu na świecie nie ma żadnego lasu, żadnego drzewa ani żadnego zwierzęcia. Program czyta dane ze standardowego wejścia i wykonuje niżej opisane polecenia. W tych poleceniach parametr `las` jest nazwą lasu, parametr `drzewo` jest nazwą drzewa, a parametr `zwierzę` jest nazwą zwierzęcia. Prawidłowe nazwy są ciągami niebiałych znaków rozszerzonego ASCII o kodach od 33 do 255. Nazwę prawidłowego polecenia i jego parametry oddzielają białe znaki (znaki ' ', '\t', '\v', '\f', '\r' w C o kodach ASCII odpowiednio 32, 9, 11, 12, 13). Każde prawidłowe polecenie pojawia się w osobnym wierszu zakończonym linuksowym znakiem przejścia do nowej linii (znak `\n` w C, kod ASCII 10). Białe znaki na początku i końcu wiersza należy ignorować.

- `ADD las` – Dodaje las.
- `ADD las drzewo` – Dodaje las i drzewo w lesie.
- `ADD las drzewo zwierzę` – Dodaje las, drzewo w lesie i zwierzę żyjące na tym drzewie.

Polecenie `ADD` dodaje tylko te byty, których brakuje. Ponowna próba dodania istniejącego już na świecie bytu jest ignorowana.

- `DEL` – Usuwa wszystkie lasy, wszystkie w nich drzewa i wszystkie zwierzęta żyjące na tych drzewach.
- `DEL las` – Usuwa las, wszystkie w nim drzewa i wszystkie zwierzęta żyjące na tych drzewach.
- `DEL las drzewo` – Usuwa drzewo w podanym lesie i wszystkie zwierzęta żyjące na tym drzewie.
- `DEL las drzewo zwierzę` – Usuwa zwierzę żyjące na danym drzewie w podanym lesie.

Polecenie `DEL` usuwa tylko te byty, które istnieją. Próba usunięcia nieistniejącego bytu jest ignorowana.

- `PRINT` – Wypisuje wszystkie lasy.
- `PRINT las` – Wypisuje wszystkie drzewa w podanym lesie.
- `PRINT las drzewo` – Wypisuje wszystkie zwierzęta żyjące na danym drzewie w podanym lesie.

Polecenie `PRINT` wypisuje nazwy bytów posortowane leksykograficznie rosnąco według rozszerzonych kodów ASCII, każdą nazwę w osobnym wierszu. Jeśli nie ma żadnego bytu do wypisania, to niczego nie wypisuje.

- `CHECK las` – Sprawdza, czy istnieje las o danej nazwie.
- `CHECK las drzewo` – Sprawdza, czy istnieje drzewo o danej nazwie w podanym lesie.
- `CHECK las drzewo zwierzę` – Sprawdza, czy na podanym drzewie w podanym lesie żyje zwierzę o danej nazwie.

W poleceniu `CHECK` można podać gwiazdkę jako wartość parametru z wyjątkiem ostatniego. Gwiazdka jest metaznakiem pasującym do dowolnej nazwy.

Ponadto:

- Puste i składające się z samych białych znaków wiersze należy ignorować.
- Wiersze rozpoczynające się znakiem `#` należy ignorować.

Informacje wypisywane przez program i obsługa błędów

Program kwituje poprawne wykonanie polecenia, wypisując informację na standardowe wyjście:

- Dla poleceń `ADD` i `DEL` wiersz ze słowem `OK`.
- Dla polecenia `CHECK` wiersz ze słowem `YES` lub `NO` zależnie od wyniku tego polecenia.
- Każdy wiersz wyjścia powinien kończyć się linuksowym znakiem przejścia do nowej linii (znak `\n` w C, kod ASCII 10). Jest to jedyny biały znak, jaki może pojawić się na wyjściu.

Program wypisuje informacje o błędach na standardowe wyjście diagnostyczne:

- Dla każdego błędnego wiersza, np. z powodu błędnej nazwy polecenia lub złej liczby parametrów, należy wypisać wiersz ze słowem `ERROR`, zakończony linuksowym znakiem końca linii (znak `\n` w C, kod ASCII 10). Jest to jedyny biały znak, jaki może pojawić się na wyjściu.

Przykładowe dane

Przykładowe dane dla programu znajdują się w załączonych plikach.

Zakończenie programu

Program kończy się po przetworzeniu wszystkich poleceń z wejścia. Program powinien wtedy zwolnić całą zaalokowaną pamięć i zakończyć się kodem 0. Awaryjne zakończenie programu, np. na skutek niemożliwości zaalokowania potrzebnej pamięci, powinno być sygnalizowane kodem 1.

Makefile

Częścią zadania jest napisanie pliku `makefile`. W wyniku wywołania polecenia `make` powinien powstać program wykonywalny `forests`. Jeśli któryś z plików źródłowych ulegnie zmianie, ponowne wpisanie `make` powinno na nowo stworzyć plik wykonywalny. Plik `makefile` powinien działać w następujący sposób:

- osobno kompiluje każdy plik `.c`,
- linkuje wszystkie pliki `.o`,
- przy zmianie w pliku `.c` lub `.h` wykonuje tylko niezbędne akcje,
- wywołanie `make clean` usuwa plik wykonywalny i dodatkowe pliki powstałe podczas kompilowania.

Docelowo pliki rozwiązania należy kompilować programem `gcc` z opcjami:

```
-Wall -Wextra -std=c11 -O2
```

Skrypt testujący

Osobną częścią zadania jest napisanie skryptu `test.sh`. Po wywołaniu

```
./test.sh prog dir
```

skrypt powinien uruchomić program `prog` dla wszystkich plików wejściowych postaci `dir/*.in`, porównać wyniki z odpowiadającymi im plikami `dir/*.out` i `dir/*.err`, a następnie wypisać, które testy zakończyły się powodzeniem, a które niepowodzeniem. Skrypt powinien akceptować parametry z dowolną ścieżką, jaką akceptuje powłoka.

Do wykrywania problemów z zarządzaniem pamięcią należy użyć programu `valgrind`.

Pozostałe wymagania

Rozwiązanie zadania powinno być napisane w języku C i korzystać z dynamicznie alokowanych struktur danych. Implementacja powinna być jak najefektywniejsza. Należy unikać zbędnego alokowania pamięci i kopiowania danych.

Kod programu powinien być podzielony na moduły.

Moduł zwykle składa się z dwóch plików, np. `x.c` i `x.h`, gdzie `x` jest nazwą modułu, implementowanej przez ten moduł struktury danych lub tp. Plik nagłówkowy `x.h` zawiera deklaracje operacji, struktur udostępnianych przez moduł `x`, a plik `x.c` – ich implementację. W pliku nagłówkowym należy umieszczać jedynie deklaracje i definicje, które są częścią interfejsu tego modułu. Wszystkie szczegóły powinny być ukryte w pliku z implementacją.

Moduł może też składać się z samego pliku nagłówkowego, jeśli udostępnia jedynie definicje stałych bądź typów, lub funkcji, które sugerujemy kompilatorowi do rozwijania w miejscu wywołania (`static inline`).

Moduł może też składać się z samego pliku z implementacją, jeśli nie udostępnia żadnego interfejsu – żadne funkcje z tego modułu nie są wywoływane z innych modułów.

Ponadto rozwiązanie powinno zawierać pliki:

- `makefile` – Patrz punkt „makefile”.
- `test.sh` – Patrz punkt „skrypt testujący”.

Rozwiązanie należy oddać jako archiwum skompresowane programem `zip` lub parą programów `tar` i `gzip`.

Punktacja

Za w pełni poprawne rozwiązanie zadania implementujące wszystkie funkcjonalności można zdobyć maksymalnie 20 punktów. Rozwiązanie niekompilujące się będzie ocenione na 0 punktów. Punkty będą odejmowane za poniższe uchybienia:

- Za każdy test, którego program nie przejdzie, traci się do 1 punktu.
- Za problemy z zarządzaniem pamięcią można stracić do 6 punktów.
- Za niezgodną ze specyfikacją strukturę plików w rozwiązaniu można stracić do 4 punktów.
- Za złą jakość kodu, brzydki styl kodowania można stracić do 4 punktów.
- Za ostrzeżenia wypisywane przez kompilator można stracić do 2 punktów.
- Za brak lub źle działający `makefile` można stracić do 2 punktów.
- Za brak skryptu testującego lub błędy w tym skrypcie można stracić do 3 punktów.

Rozwiązania należy implementować samodzielnie pod rygorem niezaliczenia przedmiotu. Zarówno korzystanie z cudzego kodu, jak i prywatne lub publiczne udostępnianie własnego kodu jest zabronione.

examples.zip

Status przesłanego zadania

Status przesłanego zadania	Nie próbowano
Stan oceniania	Nie ocenione