Duże zadanie, część 3

Celem trzeciej części zadania jest dokończenie programu kalkulatora działającego na wielomianach rzadkich wielu zmiennych. Oczekujemy poprawienia ewentualnych błędów z poprzednich części zadania oraz wprowadzenie opisanych poniżej modyfikacji i rozszerzeń. Obowiązują ustalenia z treści poprzednich części zadania i z forum dyskusyjnego dla studentów.

Konstruowanie wielomianu z tablicy jednomianów

Do stworzonej w części 1 zadania biblioteki operacji na wielomianach `poly` (pliki `poly.h` i `poly.c`) należy dodać dwie funkcje działające podobnie jak funkcja `PolyAddMonos`, ale ze zmienionym sposobem przekazywania własności tablicy jednomianów:

```
/**
 * Sumuje listę jednomianów i tworzy z nich wielomian. Przejmuje na własność
 * pamięć wskazywaną przez @p monos i jej zawartość. Może dowolnie modyfikować
 * zawartość tej pamięci. Zakładamy, że pamięć wskazywana przez @p monos
 * została zaalokowana na stercie. Jeśli @p count lub @p monos jest równe zeru
 * (NULL), tworzy wielomian tożsamościowo równy zeru.
 * @param[in] count : liczba jednomianów
 * @param[in] monos : tablica jednomianów
 * @return wielomian będący sumą jednomianów
 */
Poly PolyOwnMonos(size_t count, Mono *monos);

/**
 * Sumuje listę jednomianów i tworzy z nich wielomian. Nie modyfikuje zawartości
 * tablicy @p monos. Jeśli jest to wymagane, to wykonuje pełne kopie jednomianów
 * z tablicy @p monos. Jeśli @p count lub @p monos jest równe zeru (NULL),
 * tworzy wielomian tożsamościowo równy zeru.
 * @param[in] count : liczba jednomianów
 * @param[in] monos : tablica jednomianów
 * @return wielomian będący sumą jednomianów
 */
Poly PolyCloneMonos(size_t count, const Mono monos[]);
```

Przy implementowaniu tych funkcji należy unikać powtarzania kodu.

Użyte w poniższych przykładach makra `C` i `P` oraz funkcja `M` są zdefiniowane w testach do części 1 zadania, w pliku `poly_test.c`.

Przykład 1

Funkcja `PolyOwnMonos` przejmuje na własność zarówno zawartość tablicy `monos`, jak i pamięć zajmowaną przez tę tablicę. Wywołanie funkcji `PolyDestroy` musi zwolnić wszystkie zasoby.

```
Mono *monos = calloc(2, sizeof (Mono));
assert(monos);
monos[0] = M(P(C(-1), 1), 1);
monos[1] = M(P(C(1), 1), 2);
Poly p = PolyOwnMonos(2, monos);
PolyDestroy(&p);
```

Przykład 2

Funkcja `PolyCloneMonos` nie przejmuje na własność żadnych zasobów i nie modyfikuje zawartości tablicy `monos`. Utworzone wielomiany `p1` i `p2` są takie same. Oprócz wywołań funkcji `PolyDestroy` konieczne są

wywołania funkcji `MonoDestroy` i zwolnienie pamięci za pomocą funkcji `free`.

```
Mono *monos = calloc(2, sizeof (Mono));
assert(monos);
monos[0] = M(P(C(-1), 1), 1);
monos[1] = M(P(C(1), 1), 2);
Poly p1 = PolyCloneMonos(2, monos);
Poly p2 = PolyCloneMonos(2, monos);
PolyDestroy(&p1);
PolyDestroy(&p2);
MonoDestroy(monos + 0);
MonoDestroy(monos + 1);
free(monos);
```

Składanie wielomianów

Definiujemy operację składania wielomianów. Dany jest wielomian p oraz k wielomianów q_0, q_1, q_2, $…$, q_{k-1}. Niech l oznacza liczbę zmiennych wielomianu p i niech te zmienne są oznaczone odpowiednio x_0, x_1, x_2, $…$, x_{l-1}. Wynikiem złożenia jest wielomian $p(q_0, q_1, q_2, …)$, czyli wielomian powstający przez podstawienie w wielomianie p pod zmienną x_i wielomianu q_i dla $i = 0, 1, 2, …, \min(k, l) - 1$. Jeśli $k<l$, to pod zmienne x_k, $…$, x_{l-1} podstawiamy zera. Na przykład, jeśli $k=0$, to wynikiem złożenia jest liczba $p(0, 0, 0, …)$.

W celu realizacji operacji składanie wielomianów należy rozszerzyć bibliotekę `poly` o funkcję

```
Poly PolyCompose(const Poly *p, size_t k, const Poly q[]);
```

Do interfejsu kalkulatora należy dodać polecenie

```
COMPOSE k
```

Polecenie to zdejmuje z wierzchołka stosu najpierw wielomian `p`, a potem kolejno wielomiany `q[k - 1]`, `q[k - 2]`, $…$, `q[0]` i umieszcza na stosie wynik operacji złożenia.

Jeśli w poleceniu `COMPOSE` nie podano parametru lub jest on niepoprawny, program powinien wypisać na standardowe wyjście diagnostyczne:

```
ERROR w COMPOSE WRONG PARAMETER\n
```

Wartość parametru polecenia `COMPOSE` uznajemy za niepoprawną, jeśli jest mniejsza od `0` lub większa od `18446744073709551615`.

Jeśli na stosie jest za mało wielomianów, aby wykonać polecenie, program powinien wypisać na standardowe wyjście diagnostyczne:

```
ERROR w STACK UNDERFLOW\n
```

Jak poprzednio w obu przypadkach `w` oznacza numer wiersza, a `\n` – znak przejścia do nowego wiersza.

Przykład 1

Dla danych wejściowych:

```
(1,2)
(2,0)+(1,1)
COMPOSE 1
PRINT
(1,3)
```

```
COMPOSE 1
PRINT
```

Jako wynik działania programu powinniśmy zobaczyć:

```
(2,0)+(1,2)
(8,0)+(12,2)+(6,4)+(1,6)
```

Wyjaśnienie do przykładu:

- Pierwsze polecenie `COMPOSE` podstawia wielomian x_0^2 pod x_0 w wielomianie $\left(2+x_0\right)$, więc w jego wyniku otrzymujemy wielomian $\left(2+x_0^2\right)$.
- Drugie polecenie `COMPOSE` podstawia wielomian $\left(2+x_0^2\right)$ pod x_0 w wielomianie x_0^3, więc w jego wyniku otrzymujemy wielomian $\left(8+12x_0^2+6x_0^4+x_0^6\right)$.

Przykład 2

Dla danych wejściowych:

```
(1,4)
((1,0)+(1,1),1)
(((1,6),5),2)+((1,0)+(1,2),3)+(5,7)
COMPOSE 2
PRINT
```

Jako wynik działania programu powinniśmy zobaczyć:

```
(1,12)+((1,0)+(2,1)+(1,2),14)+(5,28)
```

Wyjaśnienie do przykładu:

Polecenie `COMPOSE` podstawia do wielomianu $p = x_2^6 x_1^5 x_0^2 + \left(1 + x_1^2\right) x_0^3 + 5 x_0^7$:

- wielomian x_0^4 pod x_0,
- wielomian $\left(1 + x_1\right) x_0$ pod x_1,
- 0 pod x_2.

W rezultacie:

- wyraz $x_2^6 x_1^5 x_0^2$ przechodzi w 0,
- wyraz $\left(1 + x_1^2\right)$ przechodzi w $\left(1 + \left(1 + 2x_1 + x_1^2\right) x_0^2\right)$,
- wyraz x_0^3 przechodzi w x_0^{12},
- wyraz $5 x_0^7$ przechodzi w $5 x_0^{28}$.

Zatem cały wielomian p przechodzi w wielomian:

$0 + \left(1 + \left(1 + 2x_1 + x_1^2\right) x_0^2\right) x_0^{12} + 5 x_0^{28} = x_0^{12} + \left(1 + 2x_1 + x_1^2\right) x_0^{14} + 5 x_0^{28}$.

Przykład 3

Dla danych wejściowych:

```
((1,0)+(1,1),1)
(1,4)
```

```
COMPOSE -1
COMPOSE 18446744073709551615
```

Jako wynik działania programu powinniśmy zobaczyć:

```
ERROR 3 COMPOSE WRONG PARAMETER
ERROR 4 STACK UNDERFLOW
```

Dokumentacja

Dodany kod należy udokumentować w formacie `doxygen`.

Modyfikacja skryptu budującego

Należy dodać możliwość utworzenia pliku wykonywalnego z testami biblioteki `poly`. Czyli na przykład po wykonaniu:

```
mkdir release
cd release
cmake ..
```

- polecenie `make` tworzy plik wykonywalny `poly` całego kalkulatora,
- polecenie `make test` tworzy plik wykonywalny `poly_test` z testami biblioteki `poly`,
- polecenie `make doc` tworzy dokumentację w formacie `doxygen`.

Funkcja `main` kalkulatora ma się znajdować w pliku `src/calc.c`. Funkcja `main` uruchamiająca testy biblioteki `poly` ma się znajdować w pliku `src/poly_test.c` – może to być plik z udostępnionymi testami do części 1 zadania i rozszerzony o własne testy. Zawartość tego pliku nie będzie oceniana.

Wskazówka: W pliku `CMakeList.txt` można dodać polecenia

```
# Wskazujemy plik wykonywalny testów biblioteki.
add_executable(test EXCLUDE_FROM_ALL ${TEST_SOURCE_FILES})
set_target_properties(test PROPERTIES OUTPUT_NAME poly_test)
```

definiując uprzednio zmienną `TEST_SOURCE_FILES`.

Wymagania

Rozwiązanie części 3 zadania powinno korzystać z własnego rozwiązania poprzednich jego części. Obowiązują wszystkie wymagania z poprzednich części zadania, jeśli nie zostały zmienione w tym dokumencie.

Uwaga: niezmiernie istotne jest, aby przestrzegać opisanej specyfikacji nazw plików.