Paradygmaty programowania

Zarządzanie efektami ubocznymi z wykorzystaniem monad

Wprowadzenie

Programowanie funkcyjne unika efektów ubocznych i zachęca do tworzenia czystych funkcji, to znaczy takich, które dla danych parametrów wejściowych zawsze zwracają te same dane. O ile wprawne posługiwanie się paradygmatem pozwala ich unikać (np. poprzez unikanie obiektów mutowalnych i zmiany globalnego stanu), niestety, w wielu praktycznych sytuacjach nie jest możliwe całkowite ich wyeliminowanie. Przykładowo, niektóre funkcje mogą zależeć od operacji wejścia/wyjścia, zależeć od czasu, od innych procesów, polegać na pseudolosowości, etc.

Jednym z podejściem do obsługi efektów ubocznych w paradygmacie funkcyjnym są *monady*. Monady są sumarycznymi strukturami danych, które pomagają w organizacji procesu strukturyzacji efektów ubocznych, a następnie odpowiedniego ich przetwarzania.

Jedna z monad `Maybe`, w podstawowym zakresie – tworzona była na liście 5. Dla przypomnienia, `Maybe` pozwalał na reprezentację informacji o wyniku obliczeń (`Just`) lub o jego braku (`Nothing`). Niniejsza lista ma za zadanie zaznajomić z tworzeniem i wykorzystywaniem bardziej złożonych monad.

Zadania można wykonać w wybranym języku programowania: Scala/OCaml.

Zadania

1. Zaimplementuj własny typ sumacyjny, który będzie miał charakter monady. Niech ten typ będzie parametryzowany względem typu `X`. Nazwij ten typ `Result`. Typ ten powinien mieć 2 składniki:
    a. `Success`, który będzie przechowywał wartość typu `X`.
    b. `Failure`, który będzie przechowywał ciąg znaków z informacją o potencjalnym błędzie.
2. Zaprojektuj i zaimplementuj trzy wybrane samodzielnie funkcje obliczeniowe, które mogą spowodować błąd. Przykładami może być:
    a. funkcja zależna od I/O (np. danych pobranych od użytkownika z wejścia standardowego, danych z pliku),

b. funkcja częściowa, tzn. taka, gdzie istnieją elementy dziedziny, które mają niezdefiniowane wartości,
c. funkcje, które mogą mieć ograniczony czas wykonania (tzw. timeout),
d. funkcje operujące na strukturach indeksowanych, przy czym indeks dostarczany jest z zewnątrz.

Niech funkcje będą wzajemnie powiązane w taki sposób, by mogły stanowić sekwencję wywołań `f1` → `f2` → `f3` przykładowo:

Funkcja	**działanie**	**Potencjalny efekt uboczny**
`f1`	pobranie ciągu znaków z klawiatury	Błąd w przypadku pustego ciągu znaków lub braku odpowiedzi przez k sekund.
`f2`	konwersja znaków na liczbę (błąd, gdy jest to niemożliwe)	Błąd w przypadku, gdy ciąg znaków nie reprezentuje dziesiętnie ani szesnastkowo liczby.
`f3`	obliczenie odwrotności	Błąd, w przypadku, gdy 0.

Zamiast wypisywać informację o błędzie, zwróć obiekt typu Result w odpowiednim przypadku. W przypadku błędu, zwróć odpowiedni komunikat o błędzie. Przetestuj funkcje na różnych przypadkach.
Zaimplementuj sekwencyjnie wywoływanie funkcji `f1` → `f2` → `f3` i wypisz wynik lub komunikat o błędzie na wyjściu standardowym.

3. Zaimplementuj funkcję wiązania do obsługi monadycznego typu Result parametryzowanego względem `X`. Funkcja powinna:
   a. przyjąć na wejściu:
      i. obiekt typu `Result[X]`
      ii. unarną funkcję odwzorowującą obiekt `X` w obiekt typu `Result[Y]`,
   i zwrócić obiekt typu `Result[Y]`,
   b. w przypadku błędu – propagowować go dalej,
   c. w przypadku sukcesu – uruchomić kolejną funkcję i zwrócić jej rezultat.
4. Zaimplementuj samodzielnie wybrany operator lewostronny infiksowy (np. `>>=`) w taki sposób, by przypisać mu funkcję wiązania monady Result.
Przetestuj funkcjonalność operatora wiązania dla sekwencyjnego wywoływania funkcji `f1` → `f2` → `f3` i porównaj z wcześniejszą implementacją z zadania 2.