13 marca 2025

Programowanie obiektowe

Lista 3.

W poniższych zadaniach należy zaprogramować w języku $C^\sharp$ pewną hierachię klas reprezentującą różnego rodzaju wyrażenia wraz z konstruktorami. Każde zadanie jest warte 8 pkt, wystarczy zrobić tylko jedno zadanie.

Uwaga 1: metody `.evaluate` czy `.execute` wymagają dodatkowego parametru: słownika, w którym przechowywane są wartości zmiennych. Możesz skorzystać wprost z dostępnej w $C^\sharp$ klasy ***Dictionary***.

Uwaga 2: podane nagłówki metod `.evaluate` czy `.execute` nie są jeszcze poprawne; konieczne jest uzupełnienie ich o deklaracje wymagane dla metod wirtualnych.

Uwaga 3: konstruując odpowiednie drzewa wyrażeń reprezentujące wyrażenia nie jest wymagane parsowanie stringów; np. dla zadania 1 można je tworzyć np. tak:

```
Expression expr = new Add(new Const(4), new Variable("x"))
```

co odpowiada wyrażeniu "$4 + x$".

Uwaga 4:
Te zadania będą kontynuowane na następnej liście zadań.

Zadanie 1
Wyrażenia arytmetyczne można reprezentować jako drzewa, gdzie w liściach pamiętane są liczby, a w węzłach symbole operacji arytmetycznych. Przyjmujemy, że wyrażenia arytmetyczne składają się z liczb, zmiennych oraz operacji dodawania i mnożenia. Zaimplementuj w $C^\sharp$ odpowiednie klasy reprezentujące węzły i liście takiego drzewa jako podklasy klasy ***Expression***.
W każdej klasie zaprogramuj metodę obliczająca wartość wyrażenia

```
public int evaluate(Slownik s);
```

Można przyjąć, że będziemy tu używali tylko wyrażenia typu `int`; ale równie dobrze można użyć `float`.
Wyrażenia z jedną zmienna możemy traktować jak funkcje; możemy więc np. obliczać pochodne. Zaprogramuj również metodę

```
public Expression derivate("x")
```

która dla danego drzewa wyrażeń (będącego funkcją) zbuduje nowe drzewo reprezentujące pochodną tej funkcji względem `"x"`. Możesz przyjąć, że algorytm nie musi sprawdzać, czy drzewo jest faktycznie funkcją.

Zadanie 2
Programy w pewnym prostym języku programowania[1] składają się z kilku podstawowych instrukcji:

- instrukcja przypisania;
- instrukcja warunkowa;
- instrukcja pętli;

[1] Jest on wzorowany na tzw. programach WHILE.

- instrukcja wypisania wartości wyrażenia na konsolę.

Dodatkowo mamy też w języku do dyspozycji *listę instrukcji*. W języku tym nie mamy deklaracji zmiennych; zmienne są tworzone w momencie pierwszego przypisania wartości.
Dla każdego rodzaju instrukcji zaprogramuj odpowiednią klasę jako podklasę ***Statement*** wraz z metodą

```
public void execute(Slownik s);
```

która będzie wykonywać tę instrukcję. Można przyjąć, że wyrażenia arytmetyczne czy logiczne (np. w instrukcji podstawienia czy w instrukcji warunkowej) są pamiętane jako obiekty klasy ***string***. Tak pamiętane wyrażenia można obliczyć np. za pomocą takiego (niezbyt eleganckiego) kawałka kodu:

```
using System.Data;

DataTable dt = new DataTable();
Dictionary<string, object> dict = new Dictionary<string, object>{
    {"x", 2},
    {"y", 3}
};

var zmienna = dt.Compute($"{dict["x"]} + {dict["y"]}", null);
Console.WriteLine(zmienna);
```

Jako przykład podaj jakiś niebanalny program, np. obliczenie n-tego wyrazu ciągu Fibonacciego.
Można przyjąć, że wyrażenia podane jako *stringi* są poprawne i nie ma konieczności obsługi potencjalnych błędów w ich ewaluacji.

Zadanie 3
Zaprogramuj klasę ***Formula*** wraz odpowiednimi podklasami, które będą reprezentować formuły rachunku zdań. Przykładowo

$\neg x \vee (y \wedge \texttt{true})$

może być przedstawione jako

```
new Or(new Not(new Zmienna("x")), new And(new Zmienna("y"), new Stala(true)))
```

Przyjmujemy, że formuły składają się ze stałych `True` i `False`, zmiennych oraz przynajmniej alternatywy, koniunkcji i negacji.
Zaprogramuj w każdej klasie metody

```
public bool oblicz(Slownik s);
```

która oblicza wartość wyrażenia; przy czym argument `zmienne` jest słownikiem, gdzie kluczami są nazwy zmiennych, a wartościami wartości tych zmiennych;
Zaprogramuj metodę

```
public Formula simplify()
```

która dla danej formuły wylicza jej uproszczenie korzystając z zależności $p \wedge \mathit{false} \equiv \mathit{false}$ czy $\mathit{false} \vee p \equiv p$.