Java programozás - 1. házi feladat

Elsőéves hallgatótársak segítése (szimbolikus deriválás) (100 p + 20 p)

Elsőéves hallgatóitársaink első két félévének jelentős része a matematikáról szól. Szeretnénk nekik segíteni a deriválás mesterségének elsajátításában, viszont a lustaság nagy úr és nem szeretnénk papíron kiszámolni a deriváltakat, sokkal jobban örülnénk, ha egy program dolgozna helyettünk (tegyük fel, hogy nincs internet, így a WolframAlpha nem elérhető, illetve nem tudjuk kezelni a MATLAB Symbolic Math Toolbox-t, stb.).
Tehát a feladat egy olyan program írása, ami szimbolikusan (nem numerikusan) oldja meg a deriváltakat, tehát képes megmondani, hogy a sin(x) függvény deriváltja a cos(x) függvény.

Ehhez először is egy absztrakt osztályra (vagy interfészre) lesz szükségünk, majd néhány, azt implementáló osztályra.

`Differentiable` absztrakt osztály vagy interfész

Ez az objektum foglalja magába az összes függvényt, amit a differenciálható objektumaink megvalósítanak.

`Constant` osztály

Ez az osztály reprezentálja a konstans kifejezéseket, ez nélkülözhetetlen lesz az együtthatók és a kitevők implementálásához (`f(x) = 3.14`).

`Identity` osztály

Ez az osztály reprezentálja az identitás függvényünket (`f(x) = x`).

`Function` osztály

Ez az osztály fogja reprezentálni a függvényeket. Minden függvénynek van egy neve (például `cos`), ami a függvényt jellemzi és egy argumentuma (ami egy `Differentiable` objektum lesz).

`Expression` osztály

Ez az osztály fogja reprezentálni a műveleteket (összeadás, kivonás, szorzás, osztás, hatványozás). Minden kifejezésnek van két `Differentiable` operandusa és egy operátora.

Követelmények

`Differentiable` absztrakt osztály vagy interfész követelményei

- `public String toString()` függvény, ami szöveggé alakítja az objektumokat.
- `public Differentiable diff() throws UnknownFunctionException, UnknownOperatorException` függvény, ami visszaadja az adott objektum deriváltját. Mivel a kimenet is `Differentiable` , így lehetőségünk van többször is deriválni kifejezéseket.

`Constant` osztály követelményei

- Öröklődik a `Differentiable` osztályból (vagy implementálja az interfészt).
- `double` mező, ami a konstans értékét tárolja.

- `public Constant(double)` konstruktor.

`Identity` osztály követelményei

- Öröklődik a `Differentiable` osztályból (vagy implementálja az interfészt).
- `public Identity()` konstruktor.

`Function` osztály követelményei

- Öröklődik a `Differentiable` osztályból (vagy implementálja az interfészt).
- `String` mező, ami a függvényt tárolja szövegesen (pl `"cos"`).
- Az alábbi függvényeket kell implementálni: cos, sin, tg, ctg, ch, sh, th, cth, ln, exp, arccos, arcsin, arctg, arcctg, arch, arsh, arth, arcth.
- `Differentiable` mező, ami az argumentumot tárolja.
- `public Function(String, Differentiable)` konstruktor.
- Implementálandó a láncszabály.

`Expression` osztály követelményei

- Öröklődik a `Differentiable` osztályból (vagy implementálja az interfészt).
- `Differentiable` mezők az operandusok tárolására.
- `char` vagy `String` mező, ami az operátort tartalmazza (összeadás, kivonás, szorzás, osztás, hatványozás).
- `public Expression(Differentiable, Differentiable, String)` konstruktor (vagy `char`).
- Implementálandó deriválási szabályok:
  - Összeg deriváltja `(f+g)'=f'+g'`
  - Különbség deriváltja `(f-g)'=f'-g'`
  - Szorzat deriváltja `(f*g)'=f'*g+f*g'`
  - Hányados deriváltja `(f/g)'=(f'*g-f*g')/g^2`
  - Hatvány deriváltja `(f^g)'=f^g(g'*ln(f)+g/f*f')`
- Kiíráskor szorzat, hányados és hatvány esetén zárójelezzük mindkét operandust (ez nyilván több zárójelet fog eredményezni a kelleténél, lásd plusz pontos feladatok).

Egyéb követelmények

- `toString()` követelmények (ne ijedjünk meg, ez csak egy rakás `if`):
  - Negatív értékű `Constant` köré írjunk zárójelet.
  - Szorzó, osztó vagy hatványozó `Expression` operandusait kiíráskor zárójelezzük, ha az adott operandusok nem `Identitiy` vagy `Constant` típusúak (ezt `getClass()`, vagy `instanceof`, vagy a derivált értéke, vagy `toString()` alapján (konstansra `Double.parseDouble(asd.toString())` kivételt dob, ha nem sikerült a parse-olás, illetve identitás `toString()` kimenete "x").
  - Összeadó és kivonó `Expression` `0` értékű (`toString()` kimenetű) `Constant` operandusait ne írjuk ki.
  - Szorzó `Expression` esetén ha valamelyik operandus `0`, akkor `0`-t adjunk vissza.
  - Szorzó `Expression` `1` értékű operandusát ne írjuk ki.
  - Osztó `Expression` `1` értékű hányadosát ne írjuk ki.
  - Osztó `Expression` esetén ha a két operandus megegyezik, akkor `1`-t írjunk ki (`toString()` alapján).
  - Hatványozó `Expression` esetén, ha a kitevő `0`, vagy az alap `1`, akkor `1`-t adjunk vissza. Ha az alap `0` és a kivető nem `0`, akkor `0`-t adjunk vissza.
- A szöveges kimenetek nem megfelelő formázásáért pontlevonás jár.
- Függvény létrehozásakor dobjunk egy `UnknownFunctionException`-t, ha nem a fent specifikált függvények közül kerül ki.
- Kifejezés létrehozásakor dobjunk egy `UnknownOperatorException`-t, ha a nem fent specifikált operátorok közül kerül ki.

- (A fenti kivételeket az egyszerűség kedvéért a `diff()` függvény továbbdobja, a tesztek kapják el őket.)
- A feladatot önállóan kell megoldani.

Példa

A kifejezés `x^x`:

- `toString()` kimenete `x^x`
- `diff()` kimenete `(x^x)*(ln(x)+1)`

Az előbbi kifejezés felturbózva `x^(3*x^2)`:

- `toString()` kimenete `x^(3.0*(x^2.0))`
- `diff()` kimenete `(x^(3.0*(x^2.0)))*((3.0*((x^2.0)*(2.0/x)))*ln(x)+(3.0*(x^2.0))/x)`

További példák a unit tesztekben láthatók. Annyit érdemes észrevenni, hogy például polinom deriváltja nem feltétlenül felismerhető ránézésre (`x^2` deriváltja `(x^2.0)*(2.0/x)`), de természetesen van lehetőség ennek szépítésére (lásd plusz pontos feladatok).

Plusz pontok

Kifejezések kiértékelése (10 p)

Annak érdekében, hogy számosítva is tudjuk az eredményeket, illetve meg tudjunk oldani például szélsőérték problémákat, szükségünk van arra, hogy a kifejezésekbe be tudjunk helyettesíteni. Figyelni kell a műveleti sorrendre is.

Kiíratás szépítése (5 p)

Zárójelezések és deriválások további egyszerűsítése a feladat, például `x^2` deriváltja a szokásos módon `2.0*x` legyen (ne `(x^2)*(2.0/x)`). Ez nem nagy kihívás, csak további egyszerű kiíratási szabályok implementálása (tehát még sok `if`).

Implementálási javaslat:

- Figyeljünk gyakorlaton, amikor a reflection-ről van szó!

LaTeX (5 p)

- `public String toLaTeX()` függvény a kifejezést LaTeX kóddá alakítja.
- `public String diffToLaTeX()` függvény a kifejezés deriváltját LaTeX kóddá alakítja.

Ebben a feladatban KÖTELEZŐ valódi LaTeX parancsokat használni, úgyhogy át kell térni a tan, cot, sinh, cosh, log, stb jelölésekre (illetve nyilván a `frac{}{}` használata is kötelező, és a cos-ra és a sin-ra is van parancs!). Nem elvárás a zárójelek méretezése, de ha valaki megpróbálná, annak ajánlott a `\qty` parancs használata (lásd `physics` csomag).

Tesztkörnyezet összeállítása (JUnit 5)

A projekt létrehozása után (aminek a neve SHIBBOLETH_hf_01) a `DifferentiableTest` fájlt húzzuk bele a projektbe. A JUnit alapértelmezetten integrálva van az IDEA-ba, viszont a projekthez hozzá kell adni. Ha rámegyünk egy kipirosozott kulcsszóra (pl importok között, vagy a tesztfüggvények előtt) akkor megjelenik a szokásos sárga vagy piros villanykörte. Arra rákattintva (vagy Alt+Enter) feljön az opció, miszerint `Add 'JUnit5.x' to classpath`. Erre

rákattintva megoldódik a probléma, már csak egy Run Configuration-t kell csinálni hozzá (jobb felső sarokban Edit Configurations > + > JUnit, majd a tesztfájl megadása). Ezután a szokásos módon lehet futtatni a projektet.

Leadási határidő: 2019. március 14. 23:59.