Tema 1 - PP

Deadline:

- 26 aprilie - soft
- 28 aprilie - hard

Responsabili:

- Tudor Cebere
- George Muraru

Scop temă:
Implementarea unui interpretor care face sinteza de tip pentru un limbaj simplist.

Specificaţiile de limbaj se vor da în continuare.
Câmpurile aflate între paranteze pătrate sunt opţionale (doar pentru specificaţiile din această pagină).

- **class <simbol_clasă> [extends <simbol_clasă_părinte>]**

Se defineşte o nouă clasa cu numele <simbol_clasă> care poate să extindă <simbol_clasă_părinte> (în cazul în care apare keyword-ul **extends**).

- **newvar <simbol_variabilă> = <simbol_clasă>** instanţiază o variabilă de tipul <simbol_clasă> - variabilele se ţin doar în clasa *Global*

- **<tip_returnat> <simbol_clasă>::<simbol_funcţie) ([tip_param_1, tip_param_2 … , tip_param_n])**

Se defineşte o nouă funcţie care aparţine clasei <simbol_clasă>.
Atenţie! Pot exista funcţii cu aceeaşi denumire în aceeaşi clasă care întorc sau primesc tipuri diferite.

Ex:
Int f::A (Double, Double)
Float f::A (Double, Double, Double)

a. Pentru început ne va trebui un mod de a reţine informaţiile specifice fiecărei clase. Definiţi ce înseamnă pentru voi "container-ul" de clasă (*ClassState*) - puteţi utiliza Map-ul[0] din Haskell - şi implementaţi următoarele funcţii:

- ***initEmptyClass*** - va întoarce un container "gol" pentru o clasă
  initEmptyMap :: ClassState
- ***insertIntoClass*** - care va introduce într-o clasă primită ca parametru un nou simbol şi "valoarea" asociată acestuia.
  insertIntoClass :: ClassState → InstrType → [String] → ClassState
  Primul parametru reprezintă container-ul clasei unde dorim să adăugam o funcţie/o variabilă.
  Al 2-lea parametru poate fi **Var** sau **Func** (prezent în schelet).
  Al 3-lea parametru va fi:
  - **<tip_returnat>:<simbol_func>:[<tip_param1, tip_param2, tip_param3, …]** - pentru o funcţie
  - **[<simbol_variabilă>, <tip_variabilă>]**

  în funcţie de parametrul al 2-lea

- **getValues** - va returna toate variabilele sau funcţiile sub o anumită formă (se află specificată în continuare).
  getValues :: ClassState → InstrType -> [[String]]

  InstrType - va fi furnizat în schelet, iar în funcţie de acesta lista de rezultate va arăta în felul următor:
  - [[**<symbol>, <tip_variabilă>**]] - dacă InstrType este *Var*
  - [[**<symbol>, <tip_returnat>, <param1>, <param2> …**]] - dacă InstrType este *Func*

b. Implementaţi funcţia **parse** cu următorul tip: String → [Instruction]
   şi funcţia **interpret** cu următorul tip: Instruction → Program → Program.

Program reprezintă baza de cunoştinţe acumulată, iar *Instruction* poate reprezenta o linie din fişier (sau orice doriţi voi).

Funcţia de parsare va primi ca argument un String şi va trebui să întoarcă o listă de "instrucţiuni", iar funcţia de interpretare va trebui să primească o "instrucţiune" (rămâne la alegerea fiecăruia cum codifică o instrucţiune) şi o bază de cunoştinţe - *program* (la începutul interpretării nu va conţine nimic - la fel rămâne la alegerea fiecăruia cum se păstrează această bază de cunoştinţe) şi va trebui să o populeze cu noua informaţie primită.

Menţiuni:

- Clasele care nu au părinţi specificaţi explicit (cu extend) vor avea ca părinte clasa "Global"
- Parsarea se face de sus în jos (liniile trebuie parsate şi interpretate în ordinea în care apar în fişier).
- Dacă o clasă extinde o clasă care nu a fost interpretată (nu există în *program*), atunci clasa va avea ca părinte clasa "Global"
- Dacă se declară o nouă **variabilă** cu un tip necunoscut, atunci se va ignora acea linie - la fel şi pentru **funcţiile** care folosesc **parametri/rezultat întors** tipuri necunoscute.

Pentru acest subpunct trebuie să mai implementaţi următoarele funcţii pentru a **valida** că parsarea şi interpretarea funcţionează corect:

- **initEmptyProgram** - va returna un container pentru *Program* gol.
  InitEmptyProgram :: Program

- **getClasses** - va returna toate clasele din program
  getClasses :: Program → [String]

- **getVars** - va returna toate numele de variabile din program
  getVars :: Program → [[String]]
  Un element din rezultat este sub urmatoarea forma:
  [<nume_variabila>, <tip_variabila]

- **getParentClass** - întoarce clasa părinte pentru o clasă dată
  getParentClass :: String → Program → String

- **getFuncsForClass** - întoarce lista de funcţii pentru o clasă (nu şi cele moştenite)
  getFuncsForClass :: String → Program → [[String]]
  Un element din rezultat este sub urmatoarea forma:
  [<nume_functie>, <tip_returnat>, <tip_param_1>, … <tip_param_n>], unde n este numărul de parametrii pentru funcţie.

Observatii:

1. Se pot scrie mai multe funcţii în fişierele sursă decât cele specificate mai sus.
2. Atenţie la spaţii - pot exista mai multe spaţii între 2 tokeni dintr-un string.

Exemplu program:

class Float
class Double
class Int

newvar a = Float
newvar b = Double
newvar c = Double

class A
Double A::plus (Int, Double)
Double A::minus (Double , Double)

class B extends A
Double B::minus(Double,Double)
Int C::plus(Int, Double)

Class C extends B
Int C::sqrt(Int)
Int C::sum(Double, Double)

c. Pentru acest subpunct trebuie să se realizeze inferența de tip pentru o expresie.
Expresia este furnizată sub forma unui datatype - un expression tree.
O expresie are următoarea formă:
expr = FCall <simbol_variabilă> <simbol_funcție> (<tip_param_1>, <tip_param_2>, ... <tip_param_n>) unde n reprezintă numărul de parametrii ai expresiei.
O expresie mai poate fi: *Va <simbol_variabilă>*, iar atunci expresia are direct tipul variabilei (bineînteles dacă aceasta există în *program*).
<tip_param_1>, <tip_param_2> ... <tip_param_n> reprezintă parametrii expresiei și pot fi la rândul lor expresii → O funcție apelată poate primi la rândul său ca argument un alt apel de funcție sau o variabilă.

Funcția trebuie să arate în felul următor:
infer :: Expr → Program → Maybe String

Paşi inferenţă:

- pentru funcţie
  - se verifică dacă există variabila din care se apelează metoda, iar în caz că nu există sinteza de tip va eşua.
  - dacă nu există metoda apelată în clasa specificată şi nici în clasele de pe lanţul de moştenire atunci sinteza de tip va eşua.
  - în caz ca există atât funcţia cât şi variabila instanţiată, iar tipurile parametrilor funcţiei nu coincid cu argumentele funcţiei atunci se continuă verificarea pe arborele de expresii.
- pentru variabilă - verificăm dacă variabila există în *program*
  - dacă există se continuă verificarea pe arborele de expresii
  - dacă nu există sinteza de tip nu reuşeste.

infer primeşte o expresie şi un *program* şi va trebui să întoarcă

- **Just <tipul_expresiei>** dacă s-a reuşit sinteza de tip
- **Nothing** - dacă nu s-a reuşit sinteza de tip

<tipul_expresiei> este reprezentat de o clasă din *program*.

Bonus:

Pentru acest subpunct va trebui să se realizeze parsarea expresiei şi realizarea inferenţei de tip.

Funcţia de *interpret* va primi acum un nou prim *keyword* (pe lângă *class* şi *newvar*) şi anume *infer* care va încerca să facă inferenţa de tip pentru o nouă variabilă.

Dacă aceasta va reuşi, se va adăuga noua informaţie în *program* (va exista o nouă variabilă cu tipul rezultat), iar în caz contrar *program-ul* rămâne la fel.

În input vor apărea şi linii precum:
infer <simbol_var> = expr

[0] https://hackage.haskell.org/package/containers-0.4.2.0/docs/Data-Map.html

Rulare checker

ghci -itester_helper
:l TestsHW1
checkAll (rulare toate testele)
check 0, 1, 2, 3 pentru rulare diferite task-uri

Punctaj:
Task a - 20p
Task b - 40p
Task c - 30p
Bonus - 20p
Lizibilitate cod + README - 10p