

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.

Specificațiile de limbaj se vor da în continuare.

Câmpurile aflate între paranteze pătrate sunt opționale (doar pentru specificațiile din această pagină).

- **class** <simbol_clasă> [extends <simbol_clasă_părinte>]
Se definește o nouă clasă cu numele <simbol_clasă> care poate să extindă <simbol_clasă_părinte> (în cazul în care apare keyword-ul **extends**).
- **newvar** <simbol_variabilă> = <simbol_clasă> instanțiază o variabilă de tipul <simbol_clasă> - variabilele se țin doar în clasa *Global*
- <tip_returnat> <simbol_clasă>::**<simbol_funcție>** ([tip_param_1, tip_param_2 ... , tip_param_n])

Se definește o nouă funcție care aparține clasei <simbol_clasă>.

Atenție! Pot exista funcții cu aceeași denumire în aceeași 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 reține informațiile specifice fiecărei clase. Definiți ce înseamnă pentru voi “container-ul” de clasă (*ClassState*) - puteți utiliza Map-ul[0] din Haskell - și implementați următoarele funcții:

- **initEmptyClass** - va întoarce un container “gol” pentru o clasă
initEmptyMap :: ClassState
- **insertIntoClass** - care va introduce într-o clasă primită ca parametru un nou simbol și “valoarea” asociată acestuia.

insertIntoClass :: ClassState → InstrType → [String] → ClassState

Primul parametru reprezintă container-ul clasei unde dorim să adăugăm o funcție/o variabilă.

Al 2-lea parametru poate fi **Var** sau **Func** (prezent în schelet).

Al 3-lea parametru va fi:

- **<simbol_func>:<tip_returnat>:[<tip_param1, tip_param2, tip_param3, ...]** - pentru o funcție
- **[<simbol_variabilă>, <tip_variabilă>]**

în funcție de parametrul al 2-lea

- **getValues** - va returna toate variabilele sau funcțiile sub o anumită formă (se află specificată în continuare).

getValues :: ClassState → InstrType → [[String]]

InstrType - va fi furnizat în schelet, iar în funcție 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. Implementați funcția **parse** cu următorul tip: String → [Instruction]
și funcția **interpret** cu următorul tip: Instruction → Program → Program.

Program reprezintă baza de cunoștințe acumulată, iar *Instruction* poate reprezenta o linie din fișier (sau orice doriți voi).

Funcția de parsare va primi ca argument un String și va trebui să întoarcă o listă de “instrucțiuni”, iar funcția de interpretare va trebui să primească o “instrucțiune” (rămâne la alegerea fiecăruia cum codifică o instrucțiune) și o bază de cunoștințe - *program* (la începutul interpretării nu va conține nimic - la fel rămâne la alegerea fiecăruia cum se păstrează această bază de cunoștințe) și va trebui să o populeze cu noua informație primită.

Mențiuni:

- Clasele care nu au părinți specificați explicit (cu `extend`) vor avea ca părinte clasa “Global”
- Parsarea se face de sus în jos (liniile trebuie parsate și interpretate în ordinea în care apar în fișier).
- 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 și pentru **funcțiile** care folosesc **parametri/rezultat întors** tipuri necunoscute.

Pentru acest subpunct trebuie să mai implementați următoarele funcții pentru a **valida** că parsarea și interpretarea funcționează 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 următoarea forma:
`[<nume_variabila>, <tip_variabila>]`
- **getParentClass** - întoarce clasa părinte pentru o clasă dată
`getParentClass :: String → Program → String`
- **getFuncsForClass** - întoarce lista de funcții pentru o clasă (nu și cele moștenite)
`getFuncsForClass :: String → Program → [[String]]`
Un element din rezultat este sub următoarea forma:
`[<nume_functie>, <tip_returnat>, <tip_param_1>, ... <tip_param_n>]`, unde *n*

este numărul de parametri pentru funcție.

Observatii:

1. Se pot scrie mai multe funcții în fișierele sursă decât cele specificate mai sus.
2. Atenție la spații - pot exista mai multe spații 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:

$expr = FCall \langle simbol_variabilă \rangle \langle simbol_funcție \rangle (\langle tip_param_1 \rangle, \langle tip_param_2 \rangle, \dots \langle tip_param_n \rangle)$ unde n reprezintă numărul de parametri ai expresiei.

O expresie mai poate fi: $Va \langle simbol_variabilă \rangle$, iar atunci expresia are direct tipul variabilei (bineînțeleas dacă aceasta există în *program*).

$\langle tip_param_1 \rangle, \langle tip_param_2 \rangle \dots \langle tip_param_n \rangle$ reprezintă parametrii expresiei și pot fi la rândul lor expresii \rightarrow 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:

$infer :: Expr \rightarrow Program \rightarrow Maybe String$

Pași inferență:

- pentru funcție
 - se verifică dacă există variabila din care se apelează metoda, iar în caz că nu există sinteza de tip va eșua.
 - dacă nu există metoda apelată în clasa specificată și nici în clasele de pe lanțul de moștenire atunci sinteza de tip va eșua.
 - în caz ca există atât funcția cât și variabila instanțiată, iar tipurile parametrilor funcției nu coincid cu argumentele funcției 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 reușeste.

infer primește o expresie și un *program* și va trebui să întoarcă

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

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

Exemplu de program pentru inferență:

```
class Double
```

```
class Float
```

```
class A
```

```
class B extends A
```

```
class C extends B
```

```
newvar a = A
```

```
newvar b = B
```

```
newvar c = C
```

```
newvar mydouble = Double
```

```
newvar myfloat = Float
```

```
Double A::func (Double, Float)
```

```
Double A::func (Float, Float)
```

```
Double B::func (Double, Double)
```

```
Float C::funcC (Double, Double)
```

1. $expr = FCall\ "a"\ "func"\ [Va\ "mydouble",\ Va\ "myfloat"]$

Inferența de tip va reuși deoarece $expr$ va avea tipul *Double* (există funcția $func$ care primește un parametru de tipul *Double* și unul de tipul *Float*).

2. $expr = FCall\ "b"\ "func"\ [Va\ "myfloat",\ Va\ "myfloat"]$

Inferența de tip va reuși deoarece $expr$ va avea tipul *Double*. Se va căuta funcția $func$ - cu parametrii de tipul *Float* și *Float*, nu va fi găsită și se va căuta în următoarea clasă pe lanțul de moștenire (clasa A) după aceeași denumire de funcție. Se va găsi funcția care întoarce *Double*.

3. $expr = FCall\ "b"\ "func"\ [Va\ "mydouble",\ FCall\ "c"\ funcC\ [Va\ "myDouble",\ FCall\ "c"\ func\ [Va\ "mydouble",\ Va\ "myfloat"]]]$

Inferența de tip va reuși deoarece:

- cel mai *nested FCall* va întoarce ca tip *Double* (se apelează $func$ aflat în clasa A)
- următorul *FCall* va apela $funcC$ și se va întoarce *Float*
- următorul *FCall* va apela $func$ (din clasa A) și va întoarce *Double*

4. $expr = FCall\ "b"\ "func"\ [Va\ "mydouble",\ Va\ "mydouble",\ Va\ "mydouble"]$

Inferența de tip nu va reuși deoarece nu există nici o funcție $func$ pe lanțul de moștenire care să aibe 3 parametrii.

5. $expr = FCall\ "b"\ "fu"\ []$

Inferența de tip nu va reuși deoarece nu există nici o funcție fu pe lanțul de moștenire.

Bonus:

Pentru acest subpunct va trebui să se realizeze parsarea expresiei și realizarea inferenței de tip.

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

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

În input vor apărea și 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 1, 2, 3, 4 pentru a rula diferite task-uri

Punctaj:gl

Task a - 20p

Task b - 40p

Task c - 30p

Bonus - 20p

Lizibilitate cod + README - 10p