Introducere în Haskell
Laborator 1

1 Pregătirea mediului de lucru.

1.1 Instalare

Instalarea platformei pe care o vom utiliza la laborator se face urmând instrucțiunile de aici: `https://www.haskell.org/platform/`. Alegeți versiunea corespunzătoare sistemului de operare și urmați cu atenție instrucțiunile.

1.2 Editor

Pentru Haskell puteți utiliza orice editor de text obișnuit. Este recomandabil să alegeți un editor cu care sunteți familiari și care pune la dispoziția utilizatorului plugin-uri pentru limbajul Haskell (care oferă colorarea sintaxei, aliniere, formatarea codului, etc.).

2 Intepretorul `ghci`

O dată cu instalarea platformei Haskell sunt instalate și două executabile pe care le vom folosi intensiv: `ghc` și `ghci`. Primul executabil (`ghc`) este un compilator pentru limbajul Haskell, iar numele său este o prescurtare pentru *Glasgow Haskell Compiler*. Al doilea executabil (`ghci`) este un interpretor pentru Haskell.

Pentru început vom utiliza Haskell în modul interactiv, adică interpretorul `ghci` pe care îl rulăm direct din linia de comandă astfel:

```
cmd > ghci
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
Prelude>
```

Imediat după executarea comenzii `ghci` este afișată versiunea curentă și pornește un prompt în care vom putea introduce comenzi noi. Spre exemplu, o primă comandă foarte utilă este cea care afișează toate comenzile pe care le putem executa:

```
cmd > :?
Commands available from the prompt:
...
```

Pentru a ieși din `ghci` vom utiliza comanda:

```
cmd > :quit
```

Putem uneori, pentru anumite comenzi să utilizăm varianta scurtă:

```
cmd> :q
```

O altă comandă foarte utilă este `:!CMD` unde `CMD` este o comandă care poate fi rulată direct în terminal. Spre exemplu, `:!ls` pe un sistem unix va afișa conținutul directorului curent. Pe Windows, același comportament îl va avea comanda `:!dir`.

Pe măsură ce vom utiliza `ghci`, vom învăța mai multe astfel de comenzi. De reținut că toate aceste comenzi sunt precedate de ":".

2.1 Evaluarea expresiilor

În `ghci` putem evalua expresii într-o manieră simplă.

Exercițiul 2.1. Evaluați în `ghci` următoarele expresii:

```
cmd> 2
cmd> 2 + 3
cmd> 2 + 3 * 5
cmd> (2 + 3) * 5
cmd> 3 / 5
cmd> 45345345346536 * 54425523454534333
cmd> 3 / 0
cmd> True
cmd> False
cmd> True && False
cmd> True || False
cmd> not True
cmd> 2 <= 3
cmd> not (2 <= 3)
cmd> (2 <= 3) || True
cmd> "aaa" == "aba"
cmd> "aba" == "aba"
cmd> "aaa" ++ "aba"
```

După cum se poate observa și din Exercițiul 2.1 sintaxa expresiilor este cea uzuală. Totuși, în Haskell, expresiile de mai sus pot fi scrise și în forma prefixată. De exemplu, expresia `2 + 3` poate fi scrisă astfel `((+) 2 3)`.

Exercițiul 2.2. Evaluați toate expresiile de mai sus în formă prefixată. Atenție la prioritățile operatorilor!

2.2 Comanda `:t`

O comandă `ghci` foarte utilă este `:t` sau `:type`. Această comandă ne permite să aflăm tipul unei expresii:

```
cmd> :t True
True ::  Bool

cmd> :t not
```

```
not ::  Bool -> Bool
```

Observați că `True` are tipul `Bool`, iar `not` are tipul `Bool -> Bool`, adică primește un argument de tipul `Bool` și returnează un rezultat de tipul `Bool`.

Exercițiul 2.3. Utilizați `:t` pentru a afla tipurile expresiilor: `True`, `False`, `True && False`, `True && (2 <= 4)`.

Exercițiul 2.4. Utilizați `:t` pentru a afla tipul expresiei: `"aaa"`. Cereți profesorului de laborator să vă explice tipul afișat.

Exercițiul 2.5. Utilizați `:t` pentru a afla tipurile expresiilor: `2`, `2 + 3`, `(+)`. Cereți profesorului de laborator să vă explice tipurile afișate.

Exercițiul 2.6. Evaluați în `ghci` expresia `not 2`. Ce obțineți?

Evaluarea expresiei `not 2` din Exercițiul 2.6 produce o eroare:

```
<interactive>:42:5:  error:
  • No instance for (Num Bool) arising from the literal ‘2’
  • In the first argument of ‘not’, namely ‘2’
    In the expression:  not 2
    In an equation for ‘it’:  it = not 2
```

Eroarea ne spune că tipul argumentului pentru `not` nu este cel așteptat, adică un argument de tip boolean.

Exercițiul 2.7. Utilizați comanda `:t` pentru a afla tipul lui `not` și apoi tipul argumentului `2`. Ce observați?

3 Funcții și apeluri de funcții.

3.1 Apeluri de funcții.

Dacă ați rezolvat Exercițiul 2.2 deja ați învățat cum se apelează funcțiile în Haskell. Operația de adunare `(+)` este o funcție. Apelarea acestei funcții se face astfel: pe prima poziție punem numele funcției, iar pe următoarele poziții se găsesc argumentele separate prin spații. Așadar, apelul este `((+) 2 3)`.

Exercițiul 3.1. În Haskell există deja predefinite funcțiile: `succ` – care calculează succesorul unui număr, `pred` – care calculează predecesorul unui număr, `max` – care calculează maximul dintre două numere, `min` – care calculează minimul dintre două numere. Utilizați comanda `:t` pentru a afla tipurile acestor funcții. Apelați toate aceste funcții în `ghci` și verificați dacă obțineți rezultatul corect.

3.2 Definirea de funcții.

Sintaxa pentru definirea funcțiilor în Haskell este foarte simplă și o vom explica pe un exemplu:

```
id x = x
```

Funcția de mai sus este funcția identitate. Numele funcției este `id`, numele argumentului este `x`, iar după simbolul `=` este corpul funcției.

Exercițiul 3.2. Scrieți funcția de mai sus în `ghci` și apelați funcția.

Funcția care calculează suma a trei numere se poate defini astfel:

```
sumThree x y z = x + y + z
```

Exercițiul 3.3. Scrieți funcția de mai sus în `ghci` și apelați funcția.

Exercițiul 3.4. Scrieți o funcție care calculează produsul a trei numere și testați funcția în `ghci`.

Deoarece este mai dificil sa editățm funcții în linia de comandă, preferăm să scriem codul în fișiere. Un fișier Haskell are de obicei extensia `.hs`.

Exercițiul 3.5. Creați un fișier pe care îl vom numi `functii.hs` și care va conține definițiile funcțiilor `id` și `sumThree` (definite mai sus).

Pentru a încărca acest fișier în `ghci`, vom utiliza următoarea linie de comandă:

```
cmd > ghci functii.hs
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
Main>
```

Alternativ, putem încărca fișierul în `ghci` folosind comanda `:l` sau `:load`:

```
cmd > ghci
GHCi, version 8.6.3:  http://www.haskell.org/ghc/ :?  for help
Prelude> :l functii.hs
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
```

După orice modificare pe care o facem în fișier, acesta trebuie reîncărcat folosind comanda `:r` sau `:reload`:

```
*Main> :r
[1 of 1] Compiling Main ( functii.hs, interpreted )
Ok, one module loaded.
```

Exercițiul 3.6. Apelați din nou funcțiile `id` și `sumThree` care sunt acum definite în fișierul `functii.hs`.

Atunci când scriem funcții în Haskell, este recomandat să scriem și tipul funcțiilor pentru a fi siguri că acestea vor fi apelate doar pe argumentele pe care noi intenționăm să le prelucrăm în funcția respectivă. Limbajul Haskell vine cu un mecanism de inferență de tipuri. De exemplu, în cazul în care nu specificăm în mod clar tipul unei funcții, acel mecanism se folosește de informațiile pe care le are în corpul funcției pentru a calcula tipul funcției.

Exercițiul 3.7. Ce tip are funcția `sumThree`? Discutați cu profesorul de laborator cum a fost inferat tipul funcției. Apelați funcția peste argumentele `3.2`, `2` și `4`.

Specificăm explicit tipul funcției `sumThree` astfel:

```
sumThree ::  Int -> Int -> Int -> Int
sumThree x y z = x + y + z
```

Exercițiul 3.8. Ce tip afișează `ghci` pentru funcția `sumThree` acum? Apelați funcția peste argumentele `3.2`, `2` și `4`. Ce s-a întâmplat?

Mai departe, definim o funcție care calculează maximul dintre două numere:

```
myMax ::  Int -> Int -> Int
myMax x y = if x <= y then y else x
```

Exercițiul 3.9. Ce tip are funcția `myMax`? Testați funcția în `ghci`.

Exercițiul 3.10. Definiți o funcție care calculează maximul dintre 3 numere întregi și testați funcția în `ghci`.

3.3 Funcții recursive.

Așa cum era de așteptat, în Haskell putem defini funcții recursive. Funcția de mai jos calculează pentru un număr dat suma numerelor naturale până la acel număr. În cazul în care argumentul este un număr negativ, funcția va returna valoarea 0.

```
mySum ::  Int -> Int
mySum x = if x <= 0 then 0 else x + mySum (x - 1)
```

Exercițiul 3.11. Testați funcția `mySum` în `ghci`.

Exercițiul 3.12. Definiți o funcție recursivă care returnează elementul de pe poziția dată ca argument din șirul lui Fibonacci.

Exercițiul 3.13. Definiți o funcție recursivă care returnează cel mai mare divizor comun a două numere.