20CYS312 -PRINCIPLE OF PROGRAMMING LANGUAGES

Date: 13-12-2024

Name: Naren S

Roll no.: CH.EN.U4CYS22034

LAB-3

1. Basic Data Types

Objective: Get familiar with basic data types like list and tuples.

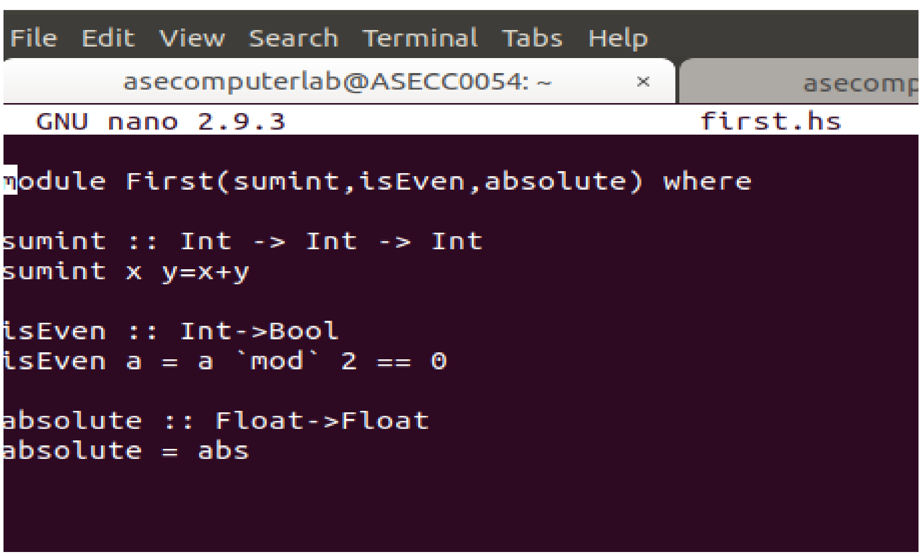
Exercise 1: **Sum of two integers**: Define a function sumint that takes two Int values and returns their sum.

Exercise 2: **Check if a number is even or odd**: Write a function isEven that takes an Int and returns a Boolean value indicating whether the number is even.

Exercise 3: **Absolute value**: Define a function absolute that takes a Float and returns its absolute value.

Code:

Using module 'First':



Output Examples:

```
*First> sumint 10 5
15
*First> sumint 10 (-5)
5
*First> sumint 10 7
17
*First> sumint (-10) (-25)
-35
*First> sumint 4.56 2432.234

<interactive>:28:8: error:
    • No instance for (Fractional Int) arising from the literal ‘4.56’
    • In the first argument of ‘sumint’, namely ‘4.56’
      In the expression: sumint 4.56 2432.234
      In an equation for ‘it’: it = sumint 4.56 2432.234
```

```
*First> sumint 7 18
25
*First> isEven 10
True
*First> isEven 0
True
*First> isEven 21
False
*First> isEven 17
False
*First>
```

```
*First> absolute 3.56
3.56
*First> absolute 3
3.0
*First> absolute (-7.5)
7.5
*First> absolute (-10.5)
10.5
*First> absolute 18
18.0
*First>
```

Explaination:

- `sumint` to add two integers.
- `isEven` to check whether a number is even.
- `absolute` to calculate the absolute value of a floating-point number.

These all are contained in the module “First”.

2.List Operations

Objective: Writing Haskell functions to perform the following tasks on lists:

<u>Exercise 1:</u> **Sum of all elements**: Define a function sumList that takes a list of integers and returns the sum of all the elements in the list.

<u>Code and Output Examples:</u>

```
ghci> let sumList :: [Int]->Int; sumList = sum
ghci> sumList [1,2,3,4,5]
15
ghci> sumList [1,-2,3,-4,5]
3
ghci> sumList [2,4,6,8]
20
ghci>
```

<u>Explaination:</u>

`sumList`: Uses Haskell's `sum` function to add up all integers in the list.

<u>Exercise 2:</u> **Filter even numbers**: Write a function filterEven that takes a list of integers and returns a list containing only the even numbers.

<u>Code and Output Examples:</u>

```
Prelude> filtereven [1,2,3,4,5]
[2,4]
Prelude> filtereven [2,4,-2,5,7.5]

<interactive>:10:22: error:
    • No instance for (Fractional Int) arising from the literal ‘7.5’
    • In the expression: 7.5
      In the first argument of ‘filtereven’, namely
        ‘[2, 4, - 2, 5, ....]’
      In the expression: filtereven [2, 4, - 2, 5, ....]
Prelude> filtereven [2,4,-2,5,7]
[2,4,-2]
Prelude> filtereven [2,4,-2,5,7,-6]
[2,4,-2,-6]
Prelude>
```

<u>Explaination:</u>

- **`filterEven`**: Uses `filter` with the previously defined `isEven` function to keep only even numbers in the list.

Exercise 3: **Reverse a list:** Define a function `reverseList` that takes a list and returns a new list with the elements in reverse order.

Code and Output Examples:

```
Prelude> let reverselist=reverse
Prelude> reverselist [1,2,3,4,5,6]
[6,5,4,3,2,1]
Prelude> reverselist [1,-5,6,-8]
[-8,6,-5,1]
Prelude>
```

Explaination:

`reverseList`: Utilizes the `reverse` function to reverse the order of elements in the list.

3. Basic Functions

Objective: Learning how to write some basic functions using recursion.

Exercise 1: **Increment each element**: Define a function incrementEach that takes a list of integers and returns a new list where each element is incremented by 1.

Code and Output Examples:

```
Prelude> let inc_each :: [Int]->[Int]; inc_each= map (+1)
Prelude> inc_each [1,2,3,4,5]
[2,3,4,5,6]
Prelude> inc_each [1,-2,-10,9,-5]
[2,-1,-9,10,-4]
Prelude>
```

Explaination:

`incrementEach`:

- Uses the map function to apply (+1) (increment by 1) to each element in the list.
- map is a higher-order function that applies a given function to all elements of a list.

Exercise 2: **Square a number**: Write a function square that takes an integer and returns its square.

Code and Output Examples:

```
Prelude> let square :: Int->Int; square x = x*x
Prelude> square 10
100
Prelude> square (-45)
2025
Prelude> square (1.5)

<interactive>:28:9: error:
    • No instance for (Fractional Int) arising from the literal ‘1.5’
    • In the first argument of ‘square’, namely ‘(1.5)’
      In the expression: square (1.5)
      In an equation for ‘it’: it = square (1.5)
Prelude>
```

Explaination:

square:

- Multiplies the input number x by itself to calculate the square.
- A straightforward implementation using the * operator.

4) Function Composition:

Objective: To write Haskell functions to perform the following tasks using **function composition**:

Exercise 1: **Compose functions to add and multiply**: Write a function `addThenMultiply` that first adds two integers and then multiplies the result by another integer. Use function composition to define this.

Code and Output Examples:

```
ghci> let addthenmul :: Int->Int->Int->Int; addthenmul x y z = (z*).(+(x+y)) $ 0
ghci> addthenmul 2 3 4
20
ghci> addthenmul (-2) 5 7
21
ghci> addthenmul 5 7 (-9)
-108
ghci>
```

Explaination:

`addThenMultiply`:

- Uses function composition `(.)` to combine addition and multiplication.
- `(z *)` creates a function that multiplies by z.
- `(+ (x + y))` adds x and y first.
- `($ 0)` applies the composed function with an initial value of `0`, ensuring the result starts from the addition.

Exercise 2: **Apply multiple transformations**: Define a function transformList that takes a list of integers and first squares each element and then adds 10 to each squared element. Use function composition to implement this.

Code and Output Examples:

```
ghci> let addthenmul :: Int->Int->Int->Int; addthenmul x y z = (z*).(+(x+y)) $ 0
ghci> addthenmul 2 3 4
20
ghci> addthenmul (-2) 5 7
21
ghci> addthenmul 5 7 (-9)
-108
ghci> let transformList :: [Int]->[Int]; transformList = map((+10).(^2))
ghci> transformList [10,9,8,7,6]
[110,91,74,59,46]
ghci> transformList [-2,-4,-6,-8]
[14,26,46,74]
ghci> transformList [18,7,10,23,0]
[334,59,110,539,10]
ghci>
```

Explaination:

`transformList`:

- Combines two operations for each list element: squaring `(^2)` and then adding `10` `(+(10))`.
- The `map` function applies this composed transformation to all elements in the list.

Conclusion:

Haskell's functional programming emphasizes expressiveness and modularity, allowing concise and reusable code. Higher-order functions and function composition provide powerful abstractions to express transformations declaratively. This makes Haskell ideal for solving problems efficiently and elegantly.