Programmierparadigmen und Compilerbau
4. Assignment Sheet

Sommersemester 2024
Prof. Visvanathan Ramesh
Gamze Akyol
Alperen Kantarci

Delivery date: 26.6.2024

Read the following information carefully!

- Group assignments are **not allowed!** However, the tasks are welcome to be solved in groups, only the tasks should be written down **individually**. Plagiarism will not be tolerated.
- Put Assignment4.hs and Assignment4_Exercise1.pdf (your solution to exercise 1) into a single .zip file on Moodle.
- The file name should correspond to the pattern `YourName_YourGroupNumber_AssignmentSheetNumber.zip`.
- You should edit todo parts or add some code depending on the exercise for each file and keep the rest of the file as it is.

1 Exercises for Assignment Sheet 4

1.1 Exercise 1 (50 points)

Consider the following simple recursion function f:

```
f x  = x + f (x-1)
```

Apply the type inference algorithm to determine the most generic type for function f.

Type inference algorithm has 5 main steps.

1. Parse program to build a parse tree
2. Assign type variables to nodes in tree
3. Generate constraints
4. Solve constraints using unification
5. Determine types of top-level declarations

At the end of the algorithm write the generic type for function f. Illustrate each step. For this exercise you will upload pdf file that contains your solution. Solution can be handwritten or digital.

1.2 Exercise 2 (50 points)

Assignment4.hs file is included in our Moodle page. You will be only working on this script.

In order to test your solutions, you need to run the main function or you need the run the following commands from your Docker environment:

```
stack runhaskell Assignment4Test.hs
```

If you want to try examples in the main function you can run the following commands:

`stack ghci Assignment4.hs`

after that you can run main or your implemented functions through ghci

```
main
```

In this assignment, you are expected to define a tiny programming language for manipulating integer lists and implement an interpreter for it using the interpreter pattern and combinators in Haskell.

The language supports the following operations:

1. Constant: Represents a constant list literal.
2. Cons: Prepends an element to the front of a list.
3. Head: Returns the first element of a list.
4. Tail: Returns the list without the first element.
5. Append: Concatenates two lists.
6. Map: Applies a function to each element of a list.
7. Filter: Keeps only the elements of a list that satisfy a predicate.

In the first part, the language representation is already given to you. You don't need to change anything here. Try to understand what each expression means and why the data types are set like this.

```
data Expr
  = ConstExpr [Int]
  | ConsExpr Int Expr
  | HeadExpr Expr
  | TailExpr Expr
  | AppendExpr Expr Expr
  | MapExpr (Int -> Int) Expr
  | FilterExpr (Int -> Bool) Expr
```

Next, you will be implementing the interpret function that evaluates an expression and returns the resulting list. Please be aware that normally HeadExpr and TailExpr should throw errors for empty lists. However, to make it simple we will be just return empty list [] instead of throwing an error message. You can check the last two examples in the main function.

```
interpret :: Expr -> [Int]
interpret = todo
```

For the second part you will implement the combinators.

There are 7 constructors that you will implement.

1. ‘myConst‘: Constructs a constant list expression.
2. ‘myCons‘: Constructs a ‘ConsExpr‘.
3. ‘myHead‘: Constructs a ‘HeadExpr‘.
4. ‘myTail‘: Constructs a ‘TailExpr‘.
5. ‘myAppend‘: Constructs an ‘AppendExpr‘.
6. ‘myMap‘: Constructs a ‘MapExpr‘.
7. ‘myFilter‘: Constructs a ‘FilterExpr‘.

```
myConst :: [Int] -> Expr
myConst = todo

myCons :: Int -> Expr -> Expr
myCons = todo

myHead :: Expr -> Expr
myHead = todo

myTail :: Expr -> Expr
myTail = todo

myAppend :: Expr -> Expr -> Expr
myAppend = todo

myMap :: (Int -> Int) -> Expr -> Expr
myMap = todo

myFilter :: (Int -> Bool) -> Expr -> Expr
myFilter = todo
```

After implementing the all, you can test your interpreter using given example main function.

```
-- Example Usage to test your implementation
main :: IO ()
main = do
  -- Construct the list [1, 2, 3, 4, 5]
  let list1 = myCons 1 (myCons 2 (myCons 3 (myCons 4 (myCons
      5 (myConst []))))))
  print (interpret list1)  -- Output: [1,2,3,4,5]

  -- Get the head and tail of the list
  print (interpret (myHead list1))  -- Output: [1]
  print (interpret (myTail list1))  -- Output: [2,3,4,5]

  -- Append two lists
  let list2 = myConst [6, 7, 8]
  let list3 = myAppend list1 list2
  print (interpret list3)  -- Output: [1,2,3,4,5,6,7,8]

  -- Map a function over a list
```

```
let list4 = myMap (* 2) list1
print (interpret list4)  -- Output: [2,4,6,8,10]

-- Filter a list based on a predicate
let list5 = myFilter even list1
print (interpret list5)  -- Output: [2,4]

-- Handle error case: taking the head of an empty list,
    will return empty list
print (interpret (myHead (myConst [])))  -- Output: []

-- Handle error case: taking the tail of an empty list,
    will return empty list
print (interpret (myTail (myConst [])))  -- Output: []
```

Normally last two examples in the main function should throw an error message. To simplify the testing, it is enough that you return an empty list for these two functions.