Nigel Salamanes - 43690599

COMP332  Assignment 2

Type analysis for the hipster language

Introduction:

The aim of the assignment was to extend the Hipster language from syntax analysis to type analysis. These stages of what a compiler does in processing source text.

**Design/Implementation:**

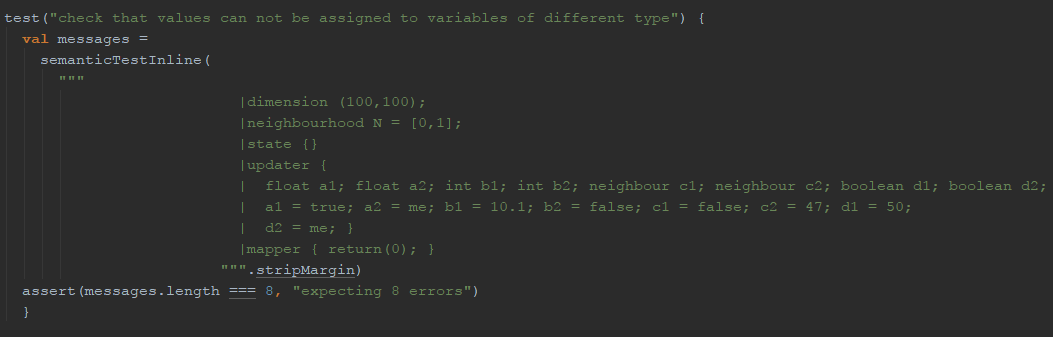
Implementing type analysis was split into two parts. There was a “tipe” and “expTipe” of an expression. “tipe” attribute infers the type of the node according to the inference rules. “expTipe” is which attribute each expression node sets with the type it may legally accept.

It was important in “tipe” that certain expressions were higher in the function than others. This precedence is because the function may be called recursively to work out its type. Apart from the given expressions, I have implemented “tipe” order in the following way UnaryExpression, EqualExpr, BinaryExpression, NeighbourExpr, FunCallExpr. The last case will return an UnknownType() however a fully implemented type analysis should never return an unknown type.

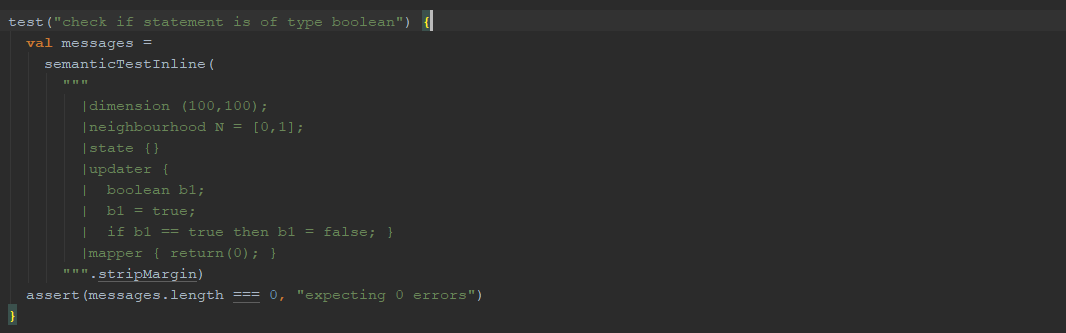
Arithmetic operation nodes use the same syntax. Because of this common structure, we could use a trait to label arithmetic operation classes. This covered plus, minus, multiply, divide, modulus, negation expressions. This heavily reduced the amount of repetitive code by instead using inline functions.

Testing:

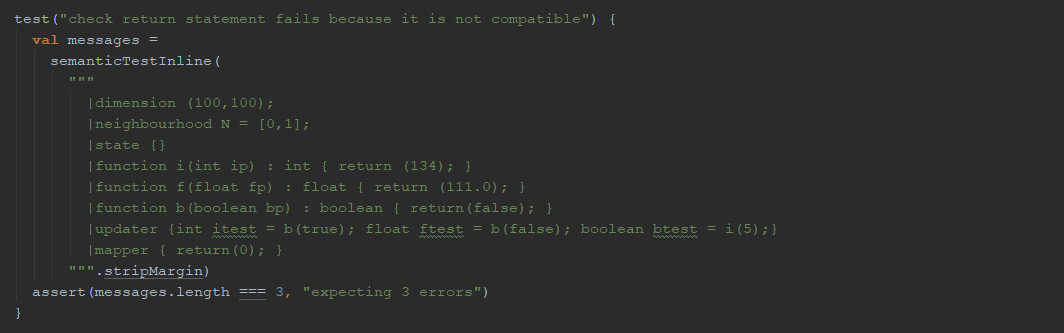
The testing covers the “expTipe” function which computes the set of expected types. Testing covers both passing and failing which ensures verification of datatypes.



This test ensures that variables can’t be assigned data of another type.



This test ensures that the statement will return a type Boolean, true or false.



This test ensures that variables can’t be assigned data from return functions with a different data type.