## DatatypesEtc

The file DataTypesEtc contains all the data types of our project. We put these in one file so that we could make the project more modular. This way all other files can just import DataTypesEtc and then they have all the data types they need. Otherwise imports would have to loop back and forward and Haskell doesn’t accept that.

## Grammar

The file Grammar contains only the grammar we used.

## Parse

The file Parse contains the tokenizer, the parser, a function to make an AST from the tree you get from the parser and a function Parse0 which is the main function of the file. It takes a string and returns the AST it would correspond to, given that there are nog errors of course.

The tokenizer first adds spaces around dots, comma’s, parenthesis, etc. so that they can be recognized as separate tokens. Then the comments are filtered out, that is a function takes elements from a list, then if it sees the word ‘btw’ it stops taking tokens but continues looking until it sees the keyword ‘.’. Then for the rest of the words it is determined what they are. First is checked if it starts with a capital letter, then it would be a function name. Then it is compared to true and false. After that we check if it may be an integer or a character (we don’t use those anymore tough). At last we check if it can be a token from the list of tokens we defined (which are the keywords) and if all else fails it is an identifier (var name).

The resulting tokenlist is put in the parser we got from Jan Kuuper. This parser builds us a tree using the grammer, which we then convert to an AST so the checker can use it.

## Checker

The Checker file contains the part where the AST is checked for type errors and scope errors. In this file, the error printing is also included. The main function is check, which takes an AST and checks it for type and scope errors. If it finds any, it will print them and throw a Haskell error so the compiling stops.

The function that checks the errors works with pattern matching so it is basically a node visitor, besides that is the function recursive so it is a synthesised attribute. The checker just runs over a list of nodes, then checks its sub trees, adds those errors to the error list, then checks the rest of its ‘brothers’ and add that to the error list too. Because the variables list is passed on in the recursion too, every time you go down a node, it is basically a new scope. Just in some cases such as program, task, when and while a special tuple is added to be able to see when the new scope starts for later checking if a variable may be re-declared.

In a declaration and task node, new variables are defined so then the function to make a new tuple for the variable list is called and the tuple is added to the variable list.

There is also a function that checks an expression and then returns the type the expression would result in. We couldn’t build this in the main checker function because this wouldn’t work with the return types. The main checker now just lets a separate function evaluate the expression so it just has to check if the types match.

## Main

The main file is actually very simple. This is the only impure file we have. It’s main function is compile, which puts the whole project together. It takes a string that represents a file path, this should be a .txt. compile reads the complete file as a string and puts it in the parse0 function from the file Parse, which gives back an AST. Then compile puts that AST into the checker and if that doesn’t throw an error it will continue with the code generation. At the end it will print the sprockel code.

In this file there is also a function that works the same with the file reading and then creates an AST from that string. Then instead of checking and generating code, it will show the tree using the ‘standard webpage 2’ from the colleges.