



For the third milestone (M3) of the compiler we aim at validating code syntactically correct code.

Once again M3 is delivered as a docker container (max 1MB), so that I may run your code on my machine, regardless of the programming language and the versions of the software you use.

You are expected to

- Write a validator for the AGUDA programming language
- Write a compiler that reads a source file (a `.agu` file) from the command line and prints the first few error messages.
- Write a short how-to report . Possible formats: `md` only.

Requirements:

- Your compiler should pass as many tests as possible. Tests are taken from <https://git.alunos.di.fc.ul.pt/tcomp000/aguda-testing>.
- Errors: your compiler must print the line and column of each error, in addition to a short description. The max number of errors printed should be parameterised, let us start with number 5.
- Valid programs print in the end `Valid!` or a similar mark

The how-to report should include:

- How to build your compiler
- How to run a particular test
- How to run the whole test suite (valid and invalid programs)
- How to interpret the testing output (how many tests passed, which failed)
- How to change the max number of errors to be printed
- How to change the base directory where to look for tests
- A brief description of how you implemented the symbol table
- A brief description of how you implemented bidirectional type checking (or why you did not follow this approach)
- If your parser does not pass all tests, explain why
- Name your report `aguda-M3.md`; place it in the top folder of your deliverable

Bidirectional type checking The expression type checker must be composed of two functions, one to synthesise the type of an expression, the other to analyse an expression against a given type. In AGUDA syntax, the functions have the signatures similar to the following, where `Ctx` is the type of symbol tables.

```
let typeof (ctx, exp) : (Ctx, Exp) -> Type = ...
let checkAgainst (ctx, exp, type) : (Ctx, Exp, Type) ->
    Unit = ...
```

The `checkAgainst` can be implemented with one single case: given `checkAgainst (ctx, exp, type)` call `typeof (ctx, exp)` to obtain `type'` and check whether `type` and `type'` are equal types.

However, in order to obtain better error messages you should strive to implement as many particular cases as possible.

For example, take the following program.

```
let _ : Unit = if true then unit else 5
```

If expression `if true then unit else 5` is validated in `typeof` mode one expects a message of the form:

```
Error: (1,16) Expected two equal types, found Unit and Int,
for expression 'if true then unit else 5'
```

Notice that the error message points to the conditional (`if`, character 16) and the whole expression is printed.

But because we know that the type of the conditional must be `Unit`, then, in `checkAgainst` mode, we get a much more precise error message:

```
Error: (1,26) Expected type Unit, found type Int,
for expression '5'
```

In this case the error message points to the integer (5, character 26) and 5 alone is printed.