The Grand JHygge end-user test

This is an end-user test for the MSc Thesis project: "Architecting and Designing a didactic compiler in Modern Functional Java for the 'hygge' programming language" by Tomas Hagenau Andersen (s233489). In this test, current and former students of the course "02247 - Compiler Construction" at the Technical University of Denmark (DTU).

In this test, the students (that's you) are going to familiarize themselves with 'JHygge', a new hygge compiler written in Java that targets the JVM, and implement a very simple language feature: the '-' subtraction operator.

Please, don't spend hours or days trying to complete this. Do as much as you can in 1 hour and leave it at that.

Estimated time: 1 hour

1. Setting up the project:

   You'll need to unzip the project somewhere on your machine.

   You'll also need to install the Java SDK version 24. You may find SDKMAN! useful: https://sdkman.io/

   Finally, you'll need an IDE or Text Editor with support for Java, Gradle and preferably ANTLR4. You can use whatever you like, but if you don't know where to begin, I'd suggest JetBrains IntelliJ community edition with the ANTLR4 plugin.

   Once you have those installed, you are ready to go!

   Have a look at the 'README.md' to see if you can compile JHygge.

   If you want to add test cases as you go through this, you can add hygge source files in the `src/test/hygge/<stage>/<pass-or-fail>` directories similar to `hyggec` that you already know.
   Estimated time: varies depending on what's already installed on your machine.

   Describe your experience setting up the project:

2. Extending the hygge grammar with ANTLR4:

   Find the `hygge.g4` grammar file in the project under `src/main/antlr/io/github/hacktheoxidation/hygge/` and update the grammar for `addExpr`.

   You will find the ANTLR4 documentation useful: https://github.com/antlr/antlr4/blob/master/doc/lexicon.md

   Estimated time: 10 minute(s)

   Describe your experience using ANTLR4 grammars:

3. Updating the hygge AST:

   For this task, you'll have to update the 'Expr'-class to support the new subtraction feature. Open `src/main/java/io/github/hacktheoxidation/ast/Expr.java` and modify the `BinaryOperator` interface with a new deriving class: `Sub`.
   You need to know how to use sealed interfaces and records in Java to complete this task.
   Estimated time: 5 minute(s)
   Describe your experience working with sealed interfaces and records to extend the hygge AST:

4. Connecting the grammar and the AST with a visitor:
   In order to make the parser generate the correct AST, the project uses an ANTLR4 Visitor to transform the AST generated by ANTLR4 to our custom 'Node' and 'Expr' types.
   Open `AntlrHyggeNodeVisitor.java` and override the method `visitArithSub()` similar to how it has been done for `visitArithAdd()`.
   If you are curious about how visitors and listeners work in ANTLR4, have a look at: https://tomassetti.me/listeners-and-visitors/
   Estimated time: 10 minute(s)
   Describe your experience working with ANTLR4 visitors:

5. Making the interpreter work with subtraction expressions:
   Many of the compiler stages in JHygge are divided into components that perform operations, such as interpretation, typechecking, etc., on features grouped by cohesion. In contrast to hyggec, this means that the interpreter, among other things, in split across multiple files. This design means that one can implement smaller parts of the language and easily swap out alternative implementations (dependency injection).
   Head over to `StandardBinaryOperatorReducer.java` and extend the public `reduce()`-method with the `Sub` case. It would be a good idea to follow the approach of making a private overload for the `reduce()`-method as well just like what's been done for `Add' and `Mult` expressions (I think you see where this is going).
   You can use the semantic rules that you know from the course to implement subtraction or come up with your own.
   Estimated time: 10 minute(s)
   Describe your experience extending the interpreter in JHygge:

6. Typechecking subtraction expressions:
   Next up is typechecking, so head over to `StandardHyggeTypeChecker.java`. Here you'll have to update the `BinaryOperator`-case in the public `typeCheck()`-method and also the private `getRelevantTypes()`- and `getResultType()`-methods along with the `BinaryOperatorType.java` enum.
   You decide want typing rules to use, but you'll likely use rules that you have already seen in the course ;)
   Estimated time: 10 minute(s)
   Describe your experience working with the JHygge typechecker:

7. Generating JVM bytecode for subtraction expressions:
   As the last part of this test, you'll need to generate JVM bytecode for your new subtraction operator feature.
   The JVM backend in JHygge uses the new Class-file API from Java 24: https://openjdk.org/jeps/484
   Open `JVMBinaryOperatorCodeGenerator.java` and extend the public `generateCode()`-method with the `Sub`-case.
   In the same file, make a private override of `generateCode()` for `Sub`-expressions similar to ones for `Add` and `Mult`.
   You'll need to call methods on the `CodeBuilder`-instance for every opcode that you want to generate.
   The `CodeBuilder` API is based on the Builder Pattern, but the object is immutable, so make sure that you bind or return the new instance.
   You'll most likely need the `isub` and `dsub` JVM bytecodes which have corresponding names on the `CodeBuilder object`.
   Estimated time: 15 minute(s)
   Describe your experience implementing code generation for the JVM using the Class-file API:

8. Closing thoughts:
   Congratulations! You have now implemented subtraction expressions in JHygge, a compiler that was completely alien to you an hour ago.
   I would like to sincerely thank you for doing this.
   As a final request, please add any further comments and closing remarks that you may have about using JHygge: