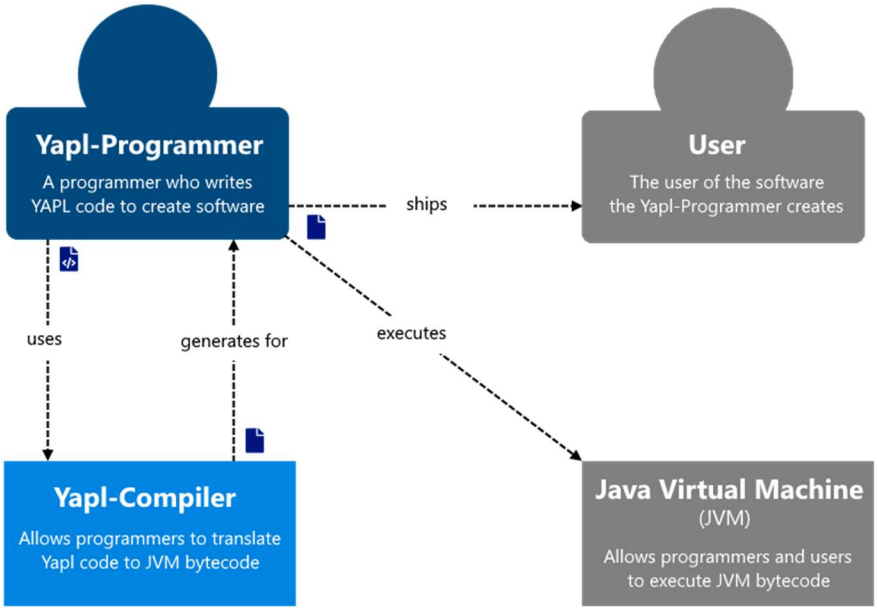
The system context

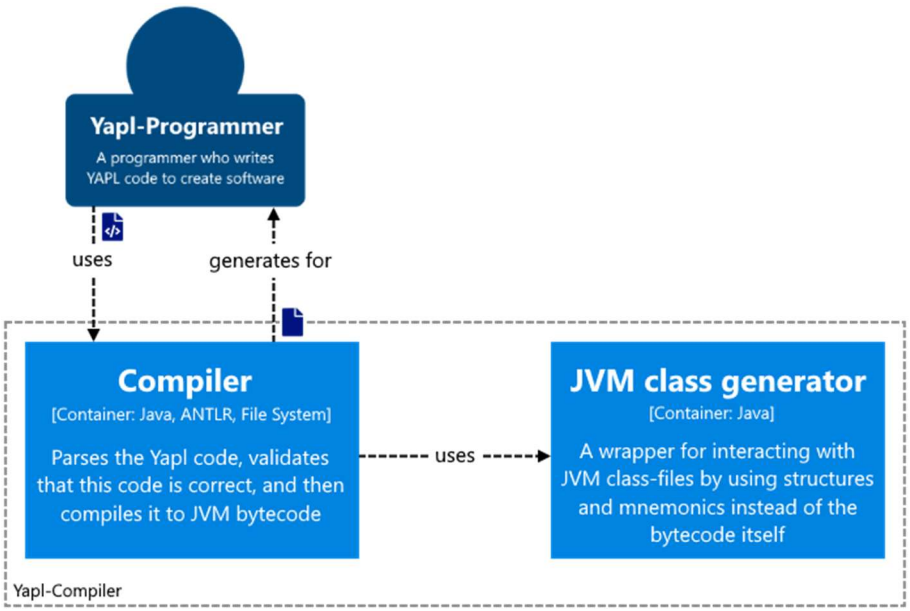
YAPL-Programmers use the YAPL-Compiler to translate their YAPL code to JVM bytecode. The YAPL-Programmers get the produced JVM bytecode and can then ship it to the users of their software, or run it on the JVM, which allows people to execute JVM bytecode.



The containers of the system

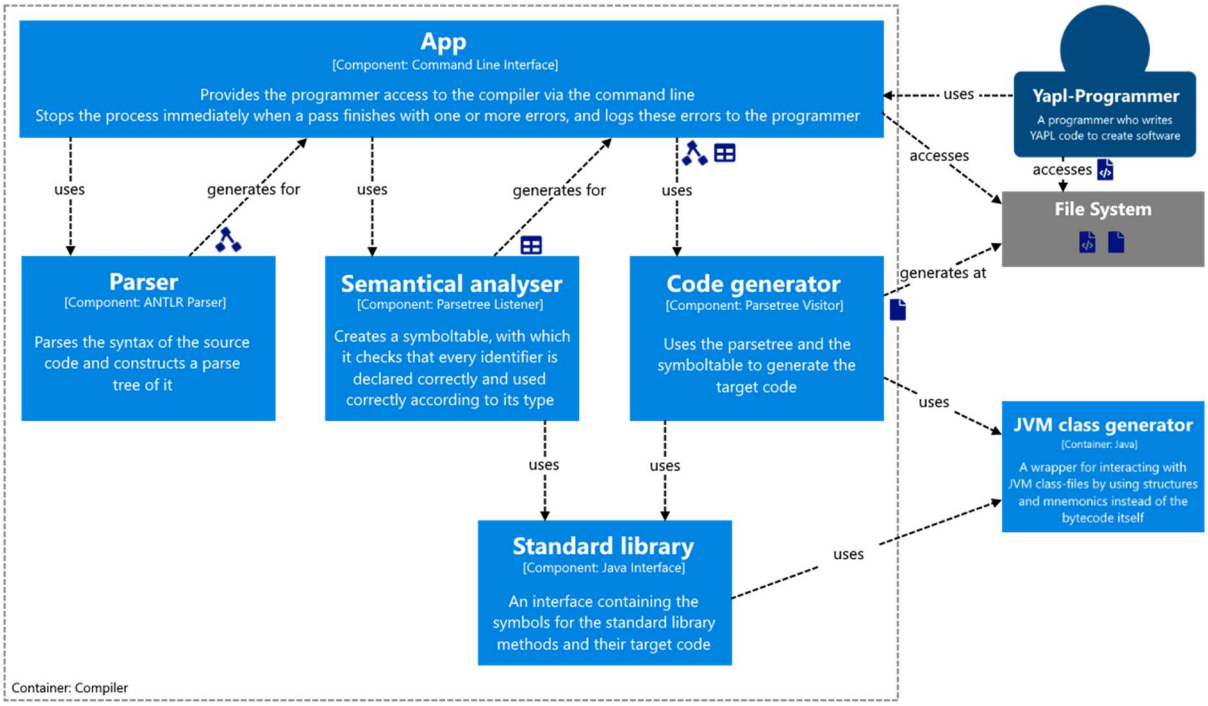
The system is made up of two containers, the main compiler and the JVM class generator. The YAPL-Programmer uses the compiler, which then parses the YAPL code, validates that this code is syntactically as well as semantically correct, and then compiles it to executable JVM bytecode. The Compiler uses the JVM class generator, a wrapper for interacting with JVM class-files which allows the programmer to use mnemonics and structures instead of the bytecode itself.

The compiler is written in Java, uses ANTLR for the parsing, and accesses the file system. The JVM class generator is written in Java as well.



The compiler components

The compilers main component is app, it is the interface the user interacts with and calls the other components in turn to create the full compilation process. When one of the components it uses/passes finishes with one or more errors it stops the process and logs these errors to the programmer. The parser parses the syntax of the source code and constructs a parse-tree, which is then passed to the semantical analyser. Creating a symbol-table and checking if identifiers are declared and used correctly is its job. Lastly, the code generator uses the parse-tree and the symbol-table to generate the target code. Since the current code generator generates JVM bytecode it uses the JVM class generator container to simplify the generation of the bytecode. The standard library provides the semantical analyser with the symbols for the predefined functions, and the code generator with the target code for these predefined functions.



The JVM class generator components

Since the project focus lies on the compiler, the JVM class generator will not be discussed in more detail. It uses the same structures and mnemonics as the official JVM specifications; therefore, it should be straight forward to use when you understand how JVM class-files are structured.

2 notes on the JVM class generator:

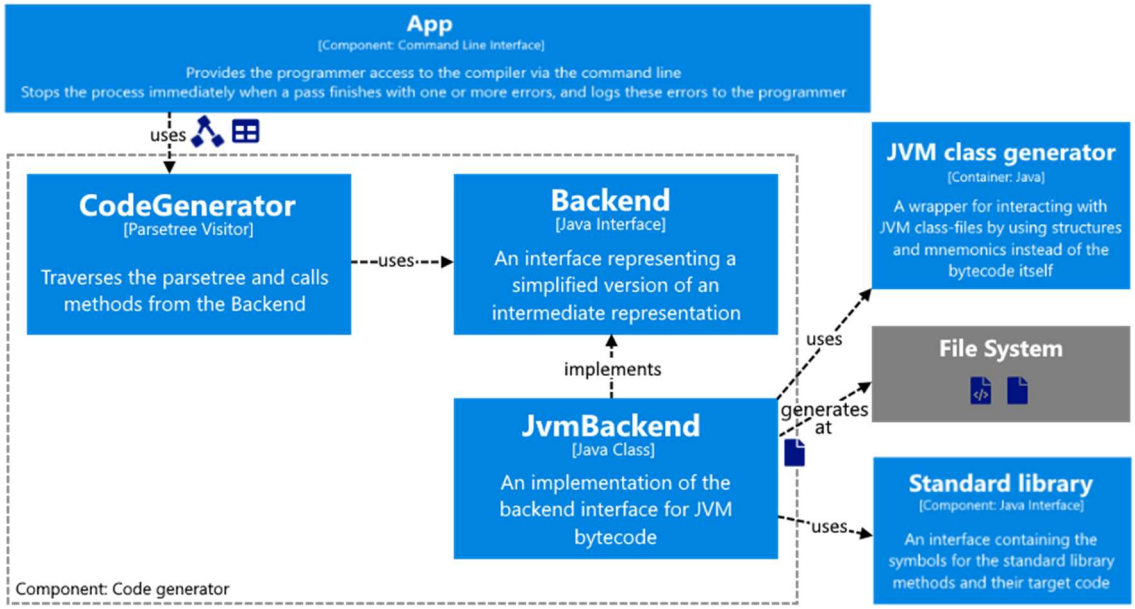
1. Not all instructions are implemented in the current version, but they can be easily added when needed.
2. The StackMapTable attribute is a very complex construction of which only the bare minimum was added to this JVM class generator.

A closer look at the code generator

The code generator in this project consists of three files. The first, CodeGenerator, traverses the parse tree and calls the Backend interface, which can be thought of as a simplified version of an IR. This interface is then implemented by the backend for a specific target code, in our case JVM bytecode. The JvmBackend then uses the JVM class generator to generate the target code and resolves calls to the standard library and generates them as such. The generated bytecode consists of two files, one for the source code and one for the standard library with the predefined functions.

Scope mapping from source code to target code:

- Program -> class
- Global variables -> static fields
- Constants -> are resolved and replaced with their values
- Records -> inner classes
- Procedures -> static methods
- Predefined procedures -> the StandardLibrary class-file with all procedures as static methods.



Developer notes & decisions

Frontend:

- ✓ ANTLR allows you to write left-recursive rules which are then automatically resolved in right-recursive rules with operator precedence. This saves the programmer some typing :). Operator precedence in a left-recursive rule for ANTLR goes from top to bottom.
- ✓ ANTLR has some built-in lexer actions that can be added to the token rules. The best example of this is the skip action which automatically skips tokens of that type. This is handy for ignoring whitespace and comments.
- ✓ ANTLR rules have operators which help with creating fuzzy parsers.
- ✓ ANTLR comes with a tool for testing the grammar, grun, which can show you token-streams and parse-trees for any input for your parser. This is a great help when traversing the parse-tree as well since it gives you more visual information of where you are at the parse-tree.
- ✓ ANTLR uses their own Adaptive LL* algorithm which works for any grammar.
- ✓ By default, ANTLR uses error recovery to try to continue the parse process. The error-messages ANTLR generates are easy to extend/customise and ANTLR gives you a lot of information on the state of the parser at the time of the error.
- ✓ A downside to using ANTLR is that the generated parser needs their ANTLR runtime.
- ✓ To traverse the parse-tree ANTLR gives you three options, interop code (which is discouraged since ANTLR v4 to make grammars more reusable), a listener pattern, and a visitor pattern. For me, the listener pattern lacked some events which made working with the parse tree a bit more annoying then needed. The visitor pattern is nice but lacks support for passing data to child-nodes via parameters. This resulted in me managing stacks with the data I otherwise would have passed as parameters. A problem both of these patterns create, is the lack of separation between the nodes and their states and actions since everything is written in one big file which implements the listener/visitor pattern.
- ✓ With ANTLR named rule children are accessed as fields and unnamed ones are accessed as methods, named alternatives of a rule are implemented via inheritance and unnamed alternatives are added as children of which only one is not null. In my opinion they should have stuck with one implementation for each of those two cases to avoid unneeded complexity. And to be honest, then a parser generator can resolve left recursion, I expect it to also be able to resolve alternatives to inheritance wherever possible, instead of only doing that when they are named.
- ✓ I changed the symbol-table implementation from a stack to a tree because in my compiler semantical analysis and code generation are done in separate passes and now, I can reuse the tree at the code generation pass.

Backend:

- ✓ Due to the JVM bytecode being object oriented and consisting of a lot of small structures, when generating the bytecode, you have a lot of tightly related structures to manage simultaneously. This bloats the generation process by such a big factor that it is its own project already.
- ✓ I really have to say that the JVM is amazingly well documented and explained in the specifications. The specifications are easy to read and give a lot of information. The only recommendation I would have for the JVM specification is sorting the instructions by opcode instead of alphabetically.
- ✓ The only poorly documented part of the JVM is the StackMapTable attribute. Unfortunately, that is also the only complicated part of the JVM. It got introduced to fix the JVMs type checking performance. Since type checking can get recursive when working with branches, the JVM speed suffered a lot of it and decided to move that to the compile time. This now means that you need to specify which types the stack-entries and locals have for every point in the bytecode where you branch. Therefore, you need to simulate the stack/locals while generating the bytecode.

- ✓ The java disassembler, javap, is like a gift made in heaven when working with JVM bytecode, it shows you the class structure the JVM recognised and the full bytecode in mnemonics with all references to the constant pool resolved and shown in a nice comment next to it.
- ✓ During this project I found out that the java compiler does not optimise the code that much since the JVM does the code optimisations JIT.
- ✓ Something that was a complete surprise to me was that all references to fields, methods, classes, and types are stored in constant utf8 strings instead of integer ids. I guess that is needed to link all the separate class-files and create stack-traces when exceptions occur.
- ✓ I ignored the existing SMBackend interface that already existed since I could not find a way to make it work with JVM bytecode.

The most challenging parts of the project:

- ✓ Working with the StackMapTable was definitely the hardest and most frustrating part of the compiler. Especially when code generation failed, and I therefore could not use javap to look at the disassembled bytecode.
- ✓ The other part of the compiler I had some struggles with is finding the right approach to working with the parse-tree ANTLR generates. Since I could not accept the idea of using ANTLRs listener/visitor pattern in my head. It just feels wrong since it ends up in big files with state managed in stacks since you cannot easily pass information between the handlers of nodes.
- ✓ Creating an interface which simulates an intermediate representation and is not dependant on the target language was another struggle, but I believe that is more of a side-effect of the previous problem.

Things that would have made my life easier during the project:

- ✓ As you can see in the diagram of the code generation, there are many layers of abstractions. Often while debugging the only thing I needed was a log with the methods that are called from the JVM class generator. It would be nice if I could tell the debugger to log all function-calls of a class or package, since adding a lot of log-statements or breakpoints is annoying.
- ✓ A way to extend objects in an existing structure/collection would have also helped a lot since then I could add the functions needed for the semantical analyses and the code generation for each node to that node itself, and simplify the state to only that part of the state each node needs. Of course, you could extend the parse-tree nodes, but then you would need to replace all tree nodes or create a new tree with the new nodes. It would be nice if you could stick your extension of the nodes to the nodes, imagine it like the original parse-tree and a tree with the extension which are then glued together and can the nodes can be accessed as one node containing the original node as well as the extension.