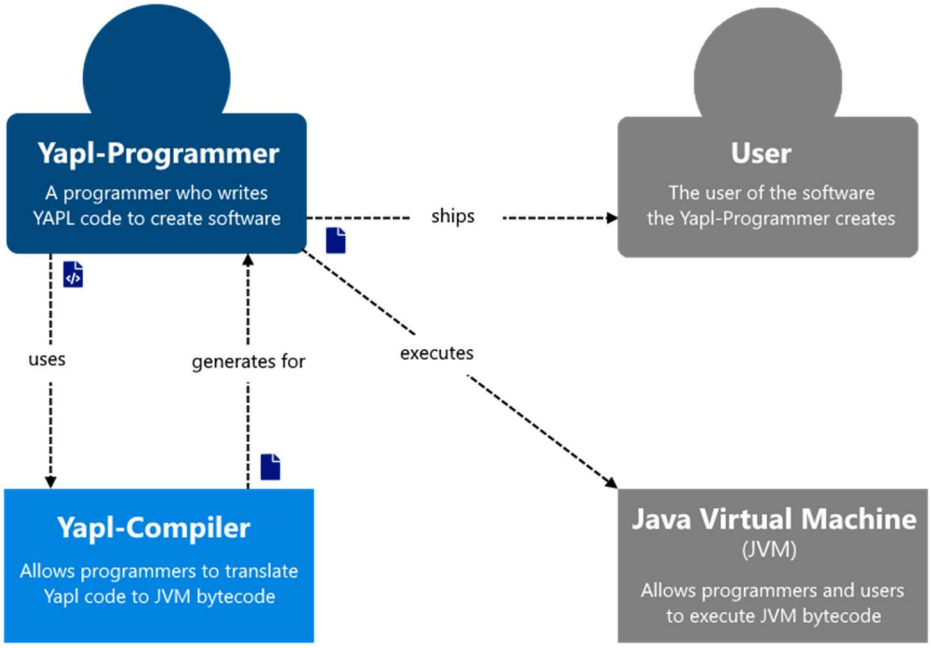
Compiler implementation using ANTLR

Internship at the University of Klagenfurt

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.



Usage

Compiles the source-file to the output-directory:

```
yapl-compiler <source-file> <output-directory>
```

Dumps all information, the compiler extracts, on symbols and expressions from the source-file:

```
yapl-compiler --symboldump <source-file>
```

Compiles the source-file to the output-directory with the given profiling options:

```
yapl-compiler <options> <output-directory>
```

Options:

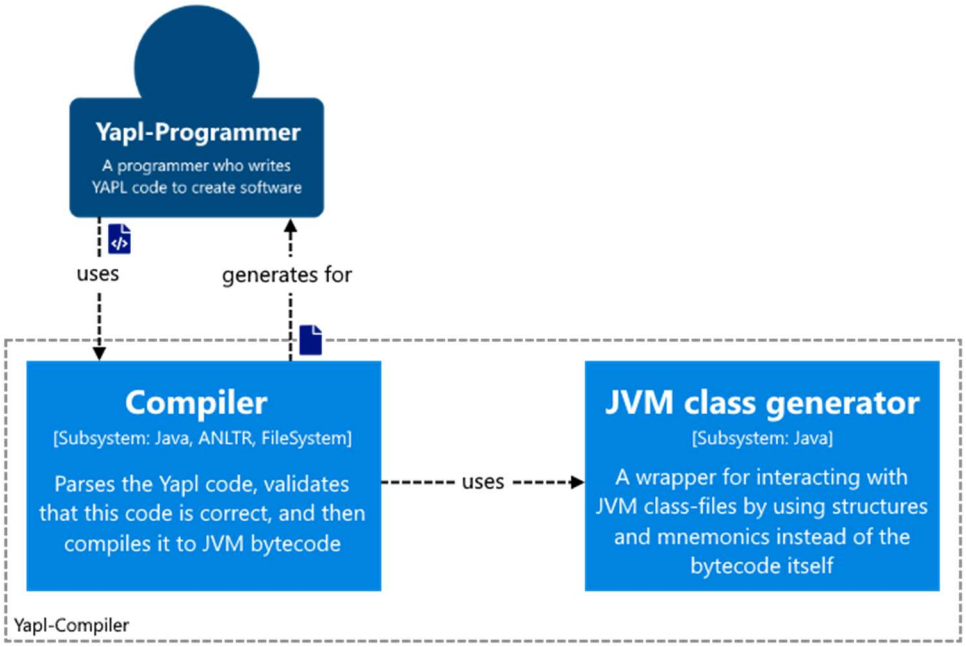
`--vardump <line1:line2:...>`	adds variable dumps before the given lines
`--watch <line1:line2:...>`	logs the expressions at the given lines
`--watch all`	logs the expressions at all lines
`--calltrace <function1:function2:...>`	logs each call to the given functions

Note: All the profiling options use the Standard Library to generate the logs, since there was not enough time to make them log into a logfile.

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 a 3-pass 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.



A quick overview of ANTLR

ANTLR generates an actual parse-tree from the source code, all the parser rules are generated as their own nodes/classes and are suffixed with `Context`. When generating the parser, ANTLR can also generate basic interfaces and classes for their listener and visitor pattern which are encouraged since ANTLR v4. This way one grammar file can be used for multiple projects and different languages.

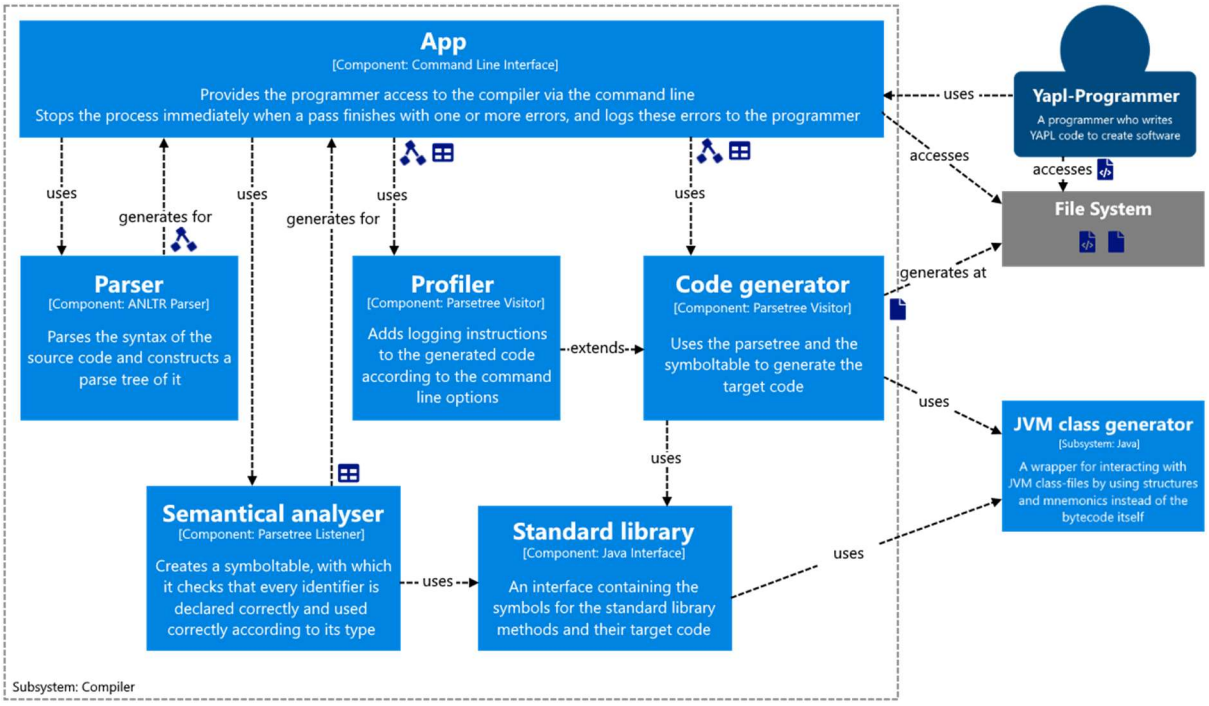
ANTLR allows you to write left-recursive rules which are automatically resolved into right-recursive rules with operator precedence (top to bottom) when the parser is generated. It also has some built-in language independent lexer actions like `skip` which automatically skips tokens of the given type. 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 way to visual the information of where you are at the parse-tree when traversing it. ANTLR has a built-in error recovery and uses their own Adaptive LL* algorithm for the parsing, which works for any grammar.

A downside to using ANTLR is that the generated parser needs their runtime. The recommended patterns also do not feel completely worked out yet since there is no easy way to pass information between methods/handlers of their interfaces in any direction. The way ANTLR generates alternatives for rules is just strange, when not named it uses a node with children of which only one is not null, and when the alternatives are named it uses inheritance. There are lots of ways to interact with (different types of) child-nodes which is more annoying than helpful. They should have sticked with one way of accessing child-nodes.

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 finishes its use/pass 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. If the programmer uses the profiling options to insert logging instructions, the profiler is used instead of the Code generator.

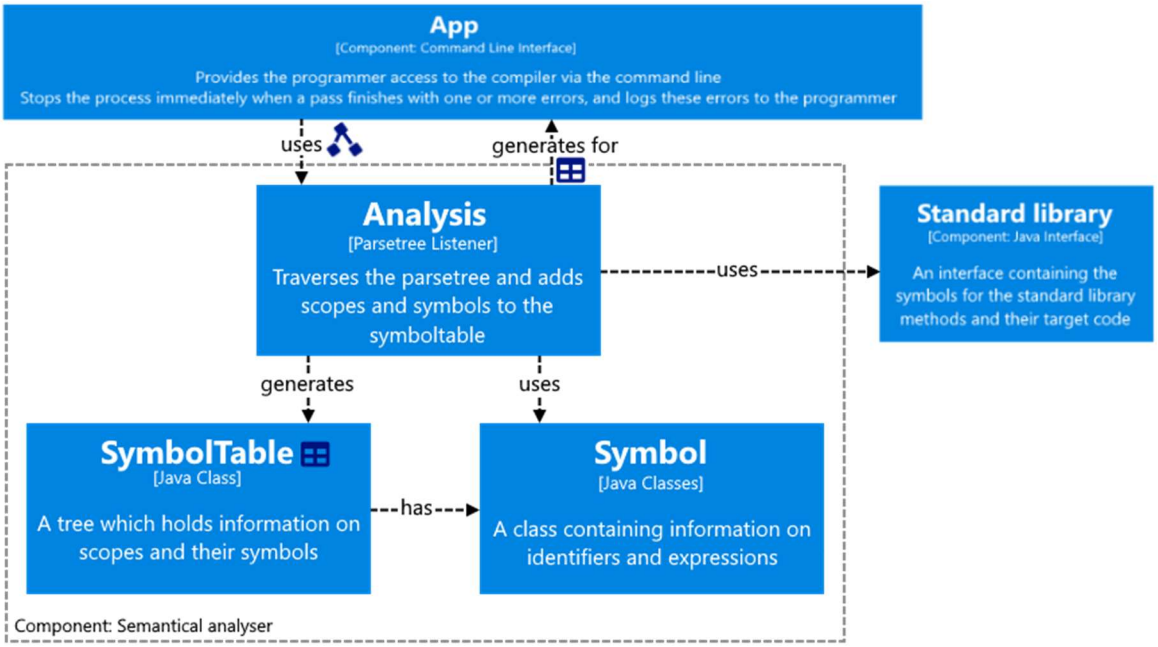
Since the current code generator generates JVM bytecode it uses the JVM class generator container to simplify the generation of the bytecode. The standard library provides the semantical analyser with the symbols for the predefined functions, and the code generator with the target code for these predefined functions.



Note: The profiler is heavily dependent on the code generator, a change in the code generator could mean a change in the profiler since the profiler has to override some of the code generators methods to add the logging.

A closer look at the Analyser and Symbol Table

The main file of the semantical analyser, Analysis, traverses the parse tree and creates a SymbolTable, with which it checks that every identifier is declared and used correctly according to its type and semantics. It also computes constant expressions and stores those in that SymbolTable. When the analyser detects an error, it is added to the list of errors and skips to the next statement to continue its checks there. The SymbolTable is a tree of scopes and their symbols. All constants, variables, functions, parameters, and records are stored as symbols. Expressions are stored as symbols as well so everything can work with symbols and no other data-classes are need for storing information on scopes, types, and values.



Notes:

The SymbolTable is implemented as a tree to extend its lifetime to all passes instead of one pass. If it were a stack, every pass would need to rebuild it since a stack can only store the symbols of the current active scopes instead of all scopes.

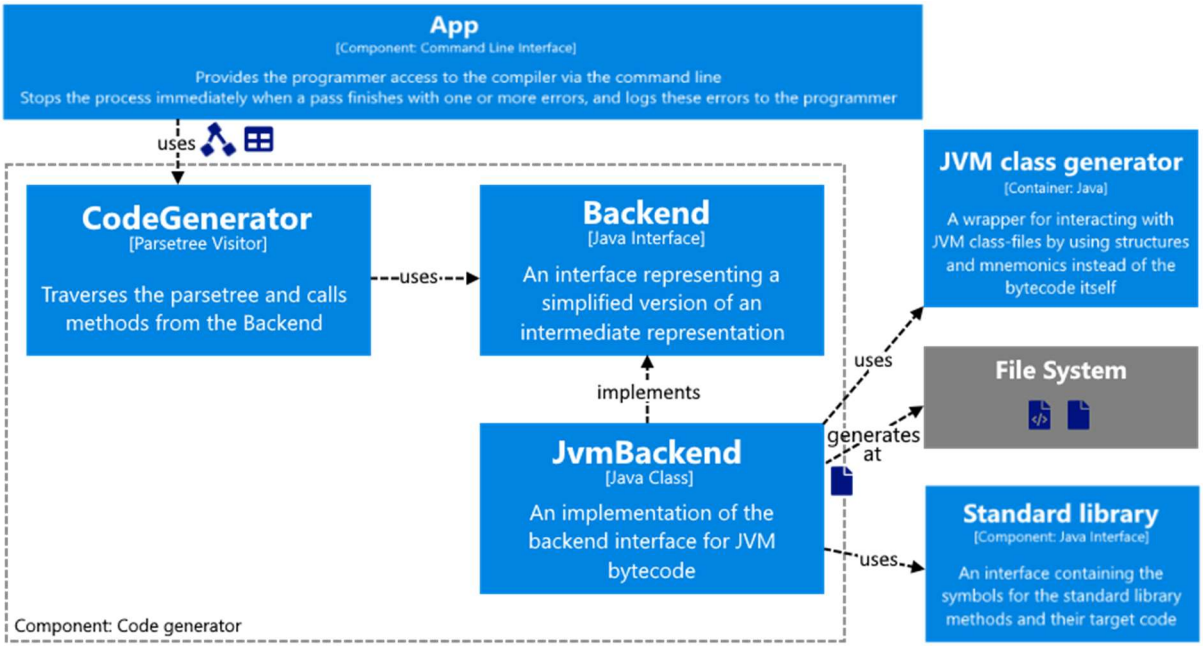
Expressions are stored as symbols as well in this implementation since they are very similar to constants except for the fact that they do not have an identifier. This simplifies the structure of the compiler since now everything works with symbols. If expressions were stored as their own attributes, multiple different but very similar data-classes would need to be managed.

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.



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.

Notes:

Not all instructions are implemented in the current version, but they can be easily added when needed.

The StackMapTable attribute is a very complex construction of which only the bare minimum was added to this JVM class generator. It is heavily dependent on the code attribute and is the thing that can be optimised the most.

Developer notes

JVM:

- ✓ Due to the JVM bytecode 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/subsystem already.
- ✓ The JVM is amazingly well documented and explained in the specifications. The specifications are easy to read and give a lot of information. A recommendation I have for the JVM specification is sorting the instructions by opcode instead of alphabetically since opcodes are grouped in operation categories.
- ✓ 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 and boils down to you needing to specify which types the stack-entries and locals have for every point in the bytecode where it branches. To get that information, you need to simulate the stack/locals while generating the bytecode.
- ✓ During this project I found out that the java compiler does not optimise the code since the JVM does the code optimisations JIT and that all references to fields, methods, classes, and types are stored in constant strings instead of integer ids. I guess that is needed to link all the separate class-files and create stack-traces when exceptions occur.

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 was 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 interface methods/handlers.
- ✓ Creating an interface which simulates an intermediate representation and is not dependant on the target language was another struggle. An important thing to note is that stack-machines and register-machines have a different order of execution which means that even with a common interface the code generator would need to be implemented multiple times to support both orders of execution.

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 call trace for the methods of 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 clutters the project.
- ✓ 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 then be accessed as one node containing the original node as well as the extension. Mainly seeing this as an interesting feature for working with big data structures and keeping the code clean.