Lecture: Compiler and Language-Processing Tools
Winter 2022/2023
Prof. Dr. Annette Bieniusa
FB Informatik, TU Kaiserslautern

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Portfolio Exam

January 25 - March 8, 2023

Marian Hristov

Eigenständigkeitserklärung

Mit Einreichen des Portfolios versichere ich, dass ich das von mir vorgelegte Portfolio selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Mir ist bekannt, dass Plagiate einen Täuschungsversuch darstellen, der dem Prüfungsausschuss gemeldet wird und im wiederholten Fall zum Ausschluss von dieser und anderen Prüfungen führen kann.

Declaration of Academic Honesty

By submitting the portfolio, I confirm that the submitted portfolio is my own work, that I have fully indicated the sources and tools used, and that I have identified and referenced the passages in the work - including tables and figures - that are taken from other works or the Internet in terms of wording or meaning. I am aware of the fact that plagiarism is an attempt to deceit which will be reported to the examination board and, if repeated, can result in exclusion from this and other examinations.

Documentation

In this part of the document the more important changes to the provided template are presented. The structure of the documentation follows the three stages of our compiler: parsing, analysis and translation, which can be found in Section 1, Section 2 and Section 3, respectively.

There are, however, some known problems in our implementation, which because of the time constraints of the examination period have not been fixed. These will be covered in Section 4.

Knowledge of the template's general structure is assumed and therefore trivial changes will not be covered here.

The new additions to the compiler are:

1. Parsing, analysis and translation of classes
    - Bonus: analysis of class inheritance
2. Parsing, analysis and translation of interfaces
3. Parsing and analysis of multi-dimensional arrays

1 Parser

In order for the compiler to be able to accept interfaces and multidimensional arrays it was for the most part only necessary to extend the grammar in the *notquitejava.cup* file, adhering to the given grammar in the task description document. This section is divided into two subsections: Interfaces and Multidimensional Arrays, each of which covers the important modifications made to the parser.

1.1 Interfaces

For the interface extension it was first necessary to add a new non-terminal symbol to the grammar, which represents the interface declaration, which can be observed in the following code snippet.

```
interfaceDecl ::=
        INTERFACE ID:name LBRACE interfaceMembers:members RBRACE
            {: RESULT = AstHelper.interfaceDecl(name, members); :}
        ;

interfaceMembers ::= // no fields allowed here
        interfaceFunctionDecl:m interfaceMembers:l // interface function-declarations can
    not have body
            {: RESULT = l; l.add(0, m); :}
        |
            {: RESULT = NQJ.FunctionDeclList(); :}
        ;

interfaceFunctionDecl ::=  // interface function-declarations are just normal function-
    declarations without a body
```

```
        type:returnType ID:name LPAREN paramList:params RPAREN SEMI
            {: RESULT = FunctionDecl(returnType, name, params, Block()); :}
        ;
```

From the code snippet we can see that an interface declaration contains interface members, which are nothing more than function declarations without a body (or implementation).

Next, in order to make interface declarations available in our parser it was needed to add the non-terminal *interface declarations* to the other global declarations of our program (e..g.. classes and functions).

```
topLevelDecl ::= classDecl:c
            {: RESULT = c; :}
        | functionDecl:f
            {: RESULT = f; :}
        | interfaceDecl:i
            {: RESULT = i; :}
        ;
```

What is left is to modify class declarations so that classes can implement interfaces. This is shown in the following lines of code:

```
classDecl ::=
        CLASS ID:name LBRACE memberDeclList:members RBRACE
            {: RESULT = AstHelper.classDecl(name, null, null, members); :}
      | CLASS ID:name EXTENDS ID:ext LBRACE memberDeclList:members RBRACE
            {: RESULT = AstHelper.classDecl(name, ext, null, members); :}
      | CLASS ID:name IMPLEMENTS interfaceImplList:impl LBRACE memberDeclList:members
    RBRACE
            {: RESULT = AstHelper.classDecl(name, null, impl, members); :}
      ;

interfaceImplList ::=
        ID:name COMMA interfaceImplList:l
            {: RESULT = l; l.add(0, name); :}
        |
        ID:name
            {: RESULT = new ArrayList<String>(); RESULT.add(name); :}
     ;
```

1.2 Multidimensional Arrays

For making multidimensional arrays available in our parser, we needed to modify the existing non-terminals to accept multiple sizes for each dimension.

```
sizesList ::=
        LBRACKET expr:e RBRACKET sizesList:l
            {:
                RESULT = l; l.addFront(e);
            :}
      |
            {: RESULT = NQJ.ExprList(); :}
      ;
```

```
expr2 ::=
        NEW baseType:t sizesList:l brackets:b
            {: RESULT = AstHelper.newArray(t, l, b); :}
      | NEW ID:t LBRACKET sizesList:l RBRACKET brackets:b
            {: RESULT = AstHelper.newArray(TypeInterfaceOrClass(t), l, b); :}
```

Other than the modifications in the *.cup* file, we made some necessary adjustments in *AstHelper.java*.

```
    public static NQJExpr newArray(NQJType t, NQJExprList sizes, int dimensions) {
        for (int i = 0; i < dimensions; i++) {
            t = NQJ.TypeArray(t, NQJ.Number(0));
        }

        for (int i = sizes.size() - 1; i > 0; i--) {
            t = NQJ.TypeArray(t, sizes.get(i).copy());
        }
        return NQJ.NewArray(t, sizes.get(0).copy());
    }
```

Here the multidimensional arrays are created using the generated *NQJ* classes, iterating through the given dimensions. In order to express each dimension's size, we added a new field in the *NewArray* object, by modifying the *notquitejava.ast* file. As the translation of multidimensional arrays was not finished at the time of writing, it is possible that the order of the created arrays' dimensions is inverted. This, however, does not put the analysis phase in jeopardy as it is consistent.

2 Analysis

As interfaces and classes are quite intertwined in this phase, this section we will separate into subsections, each of which covering the changes to a whole file (or class).

2.1 ClassType.java

This class is our internal representation of classes and is a subtype of the class *Type*. It has the following fields:

- *String* **name**: the name of the class.
- *ClassType* **superClass**: the direct super-class of the class.
- *List<InterfaceType>* **implementsInterface**: the interfaces it implements.
- *Map<String, NQJVarDecl>* **fields**: the fields of the class.
- *Map<String, NQJFunctionDecl>* **methods**: the methods of the class.

These fields are quite self-explanatory, having in mind that identifiers (e.g. class names, method names) are unique in our compiler.

Apart from some "getter" and "setter" methods, there is one noteworthy method, which overrides its super-class' one, namely *isSubTypeOf(ClassType **other**)*. This method is important in

the analysis phase whenever we need to check two objects' subtype relation (e.g. assignments). There are generally two cases of interest, depending on the argument's type **other**. Its type is either an interface or another class. In the case it is an interface, we check whether interface **other** is contained in the list of interfaces our current class (**this** object) implements. If the argument's type is a class, then we recursively check in the ancestor tree of the current class whether the argument is present or not. This can be observed in the code snippet below.

```
/**
 * Recursively checks whether the current class's parent implements the parameter
class.
 */
private boolean checkIfOtherIsParent(ClassType other) {
    ClassType parent = this.superClass;
    if (parent != null) {
        if (parent == other) {
            return true;
        } else {
            return parent.checkIfOtherIsParent(other);
        }
    } else {
        return false;
    }
}
```

2.2 InterfaceType.java

Similar to the class *ClassType* presented in Section 2.1, *InterfaceType* is our internal representation of interfaces. The class is a subtype of *Type* and contains the following fields:

- *String* **name**: the name of the class.
- *ClassType* **classImplementation**: the interface object's current class implementation.
- *List<InterfaceType>* **implementsInterface**: the interfaces it implements.
- *Map<String, NQJFunctionDecl>* **functions**: the methods of the interface.

Notably, as we will see later when we observe the changes made to the analysis of assignments, we save the interface object's class implementation in the field **classImplementation**. This field is later used in the translation phase to create the corresponding class object.

The *isSubTypeOf(ClassType **other**)* function here is quite simple. As we do not consider sub- or super-interfaces in our compiler, the only comparison to be done is to check if **other** is the same interface as the current one (meaning both have the same name).

2.3 NameTable.java

After we have covered the internal representation of interfaces and classes, we can proceed to one of the central files of the analysis phase, namely NameTable.java, in which most of the type-checking and analysis operations of classes and interfaces are being done.

Let us start with the necessary data structures, which we have defined for our needs, visible in the code snippet below. The comments in the code explain each of the data structure's use-case.

```
/**
 * A cache that maps names of classes to their respective ClassType objects.
 */
private final Map<String, ClassType> classTypes = new HashMap<>(); // similar to
arrayTypes map

/**
 * A cache that maps names of interfaces to their respective InterfaceType objects.
 */
// similar to classTypes map
private final Map<String, InterfaceType> interfaceTypes = new HashMap<>();
/**
 * A map of all globally defined classes.
 * Key: class name (string), Value: declaration (NQJClassDecl).
 */
private final Map<String, NQJClassDecl> globalClasses = new HashMap<>();

/**
 * A map of all globally defined interfaces.
 * Key: interface name (string), Value: interface declaration (NQJClassDecl)
 */
private final Map<String, NQJInterfaceDecl> globalInterfaces = new HashMap<>();
```

Now we will look at how these data structures are being filled. The entry point is the constructor of the *NameTable* object. The idea was to first fill the maps, storing the *NQJ* classes, named **globalClasses** and **globalInterfaces** and then use these to create their internal representations (*ClassType* and *InterfaceType*) and store them into the maps **classTypes** and **interfaceTypes**. One of the reasons why this was done is that it enables us to check whether class identifiers and interface identifiers are unique when filling the *NQJ* maps. Of course, we here also do not allow a class to have the same name as an interface and vice versa.

The functions *createAndAddClassType()* and *createAndAddInterfaceType()* are responsible for the creation of the internal types. Let us take a closer look at both. In *createAndAddInterfaceType()* we create an *InterfaceType* object from an *NQJInterfaceDecl* object and add it to the *interfaceTypes* map. Additionally, we check if function names are unique in the current interface declaration and throw errors accordingly. The functions *createAndAddClassType()* is a bit more sophisticated, but follows the same idea as in the interfaces' case. We create a *ClassType* object, throwing errors when we encounter some problems in the class declaration. The class' methods are not being verified here as we get this more or less "for free" from the already existing procedure in Analysis.java, which will be covered later. Errors are thrown here in the following cases:

- Field names are not unique.
- Method names are not unique
- An interface that the class implements is not defined.
- The class does not implement all functions defined its interfaces.
- The class' super-class is not defined.

- An inheritance cycle was detected.

Since if the super-class of the current class has not yet been created at the point of construction, we recursively create the super-class first (along with its whole ancestor tree). By passing a list of classes that represents the inheritance tree, we can check for inheritance cycles quite easily. As this is not part of the examination criteria and was done just for fun, we will skip further explanations here.

2.4 Analysis.java

This is the central entry point of the analysis phase. The class creates the *NameTable* object and using the visitor paradigm starts "visiting" each component of the program that is to be analyzed. The points of interest for us are the visitor methods for the classes *NQJInterfaceDecl* and *NQJClassDecl*. Let us now examine these three visitor methods.

The visitor method for *NQJInterfaceDecl* objects' job is to verify the method declarations that an interface contains. This is done with the already available in the template method that verifies global functions. This is possible as interface methods do not contain an implementation (i.e. their bodies are empty), skipping most of the existing computations and type checking.

In the case of *NQJClassDecl*, its visitor method's idea is similar to the visitor for interfaces, with the exception that the class' fields are added to the context data structure before verifying its methods.

As at the point of parsing we can not distinguish between an identifier that is a class and an identifier that is an interface, we additionally created a generated class called *TypeInterfaceOrClass*, which expresses this intuition. This class is then used in the function *Analysis.type(NQJType* ***type****)* to retrieve either of those types.

2.5 ExprChecker.java

This class is used to type-check all expressions by using the matcher paradigm. The necessary adjustments to this file were in the matcher methods for method calls, class field accesses, new object creations, and new array creations. An underlying commonality between these methods is the fallback behaviour, which returns a **Type.ANY** whenever an error is encountered in order to continue finding as much errors as possible. Let us inspect each of these methods closer.

The matcher method for method calls can be divided into two parts: receiver checking and method checking. First, the receiver of the called method is being verified. This means that the type-checker searches for the interface or the class declaration of the receiver. If it does not succeed, an error is thrown. If it succeeds in finding the interface or class, the type-checker searches for a method declaration in the object that corresponds to the given method name and throws an error if necessary. Once it succeeds, the type-checker continues with the checking of the called method itself. Here, some already existing code is used to verify the function call arguments, which logs errors whenever the arguments and the parameters do not match.

In the matcher for field accesses, we first verify that the receiver of the operation is a class and log an error if it is not. Then, we check if the class really contains such a field in its declaration.

For the creation of new objects we only need to verify that a corresponding class declaration exists.

We modified the matcher for new array creations to analyze multidimensional arrays. The new addition is an iterative loop that verifies that each dimension's length, which is encoded as an expression, is an integer.

3 Translation

As the translation of multidimensional arrays was not finished during the examination period and at the point of writing, only the translation of classes and interfaces will be presented here.

3.1 Classes

3.1.1 Class Translation and Default Constructor

The entry point for the translation of classes is the function *translate()* in the file Translator.java. We translate all classes before everything else (e.g. global functions, main function), so that classes are known and ready to be used. The translation of classes consits of three main parts, which can be observed in the following code snippet:

```
    private void translateClasses() {
        NQJClassDeclList classDecls =  javaProg.getClassDecls();
        for (NQJClassDecl classDecl : classDecls) {
            initClass(classDecl);
        }

        for (NQJClassDecl classDecl : classDecls) {
            initClassMethods(classDecl);
        }

        for (NQJClassDecl classDecl : classDecls) {
            // Don't move this inside the initClass() loop.
            // This ensures that all classes are known
            // before e.g. creating methods that have classes as params
            translateClass(classDecl);
        }
    }
```

The method *initClass()* is responsible for the initialization and storing of its corresponding *TypeStruct* object. This is done before the actual translation, since we want to make all classes known beforehand in order to avoid the problem of translating a class, which has an unknown class as a field. Next, for the same reason we initialize all structures, required for the methods of a class, so that similar errors are avoided. After that, the *translateClass()* method is called, where the main translation of a class happens. In this function, first the V-Table is initialized and filled, along with all its corresponding internal datastructures. Then, we retrieve the already initialized. but empty *TypeStruct* object and fill its fields, the first one being the pointer to the V-Table. This can be obseved in the code snippet below:

```
        TypeStruct classStruct = classStructs.get(className);
        // note: key is the name of the class STRUCT, not declaration!
        vTableStructs.put(classStruct.getName(), vTable);
```

```
        // first field is always filled with the pointer to the vTable
        classStruct.getFields().add(StructField(TypePointer(vTable), "vTable"));

        // add fields to struct
        for (final NQJVarDecl fieldDecl : classDecl.getFields()) {
            classStruct.getFields().add(
                    StructField(
                            translateType(fieldDecl.getType()),
                            fieldDecl.getName()
                    )
            );
        }
```

After we have filled the class' *TypeStruct*, we translate all class methods with the already existing code that translates global functions, with a minor modification to skip the translation of the V-Table pointer. Lastly, the default constructor for the class is generated, which allocates memory for the class and provides default values for its fields.

3.1.2 Field Access in Translated Classes

The field access matcher function can be found in ExprLValue.java. Its corresponding method retrieves the class' *TypeStruct* object that was created in Section 3.1.1 and searches for the field with the given name. This can be seen in the code snippet below.

```
        // In case of inheritance it is necessary
        // to start the search from the bottom (i.e. non-inherited fields).
        // For future convenience, we will do this here too.
        int index = -1; // the position of the field in the struct.
        for (int i = classStruct.getFields().size() - 1; i > 0; i--) {
            StructField field = classStruct.getFields().get(i);
            if (field.getName().equals(fieldName)) {
                // field was found in class struct
                index = i;
                break;
            }
        }
```

Since the original idea was to also handle classes with inheritance, we start the search for the field from the end of the field list in order to handle field shadowing. Nonetheless, as we decided to not implement inheritance because of time constraints, the order of iteration does not matter here.

It is guaranteed by our analysis phase that the field will be present in the class and so no additional error handling is necessary. Therefore, a reference to a temporary variable is returned, which contains the address to the field that is to be accessed.

In addition to the change that was made in the corresponding matcher for field accesses, we modified the matcher for the variable usage as well. This enables the user to use a field of a class in any of its methods, without having to always use the keyword "this" as receiver.

3.1.3 Method Calls of Translated Classes

Method calls are handled in the matcher function *case_MethodCall(NQJMethodCall* ***e****)* of the file ExprRValue.java. Let us examine its workflow, which can be separated into three parts. The first part finds the pointer to the method that is to be called by loading it from the V-Table of the class. The second part constructs the required parameters for the method call, including the "this" object as the first parameter. The third part of the matcher function is the action of calling the method, found in part one, with the parameters, constructed in part two.

3.2 Interfaces

The analysis phase should guarantee that each interface object in the program is used correctly. Therefore, in the translation phase it is only a matter of finding and, if required, translating the class, with which the interface variable is initialized. For this, it was necessary to pass some additional information from the analysis phase to the translation phase (file Analysis.java). What we decided on doing is for each assignment to an interface variable to save a reference to the class it is assigned to in the *InterfaceType* object. This can be observed below:

```
    public void visit(NQJStmtAssign stmtAssign) {
        Type lt = checkExpr(ctxt.peek(), stmtAssign.getAddress());
        Type rt = checkExpr(ctxt.peek(), stmtAssign.getValue());
        if (!rt.isSubtypeOf(lt)) {
            addError(stmtAssign.getValue(), "Cannot␣assign␣value␣of␣type␣" + rt
                    + "␣to␣" + lt + ".");
        } else {
            if (lt instanceof InterfaceType && rt instanceof ClassType) {
                // provide data for translation phase
                InterfaceType ltInterface = (InterfaceType) lt;
                ClassType rtClass = (ClassType) rt;
                ltInterface.setClassImplementation(rtClass);
            }
        }
    }
```

The reference can then be used in the translator to fetch the class that is assigned to the interface object.

```
            } else if (t instanceof InterfaceType) {
                InterfaceType it = (InterfaceType) t;
                result = TypePointer(classStructs.get(it.getClassImplementation().getName
    ()));
            } else {
```

4 Known Problems

As the time we had on our disposal was in no means sufficient to build a perfect compiler, this project is not devoid of errors and mistakes. Moreover, it can be viewed as a first iteration, in which we get a better picture of what the more difficult parts of building a compiler are, which

modeling abstractions work and which do not. In this section, some known problems of our compiler are listed.

- The translation of multidimensional arrays has not been implemented
- Polymorphism of interface objects does not work (see failing tests [2])
- Translator has issues with interface objects as parameters to a function when no prior assignment of the class object to an interface variable exists (see failing tests [2])

Reflection

Understanding how a compiler works has been one of my goals since the beginning of my path as a computer science student and future software engineer. However, because of the university's program structure, it was not possible for me to take the course until the start of my master's degree. Before taking the course, I had been in contact with people, who I know to have knowledge in the topic of compilers. Most recommended the notorious book by Aho et. al called "Compilers: Principles, Techniques, and Tools" [1], which is commonly known as the "dragon book" (for having a dragon on its cover). Reading the first few chapters of the book gave me an idea of what to expect from compilers, and it only strengthened my position that this topic is one that I would really like to deepen my knowledge in. And so I waited until the first year of my master's to finally take the course that I wanted to do for the majority of my bachelor's degree.

On the first lecture of the course, I learned that not only will I understand in due time (and effort) how compilers work, but that I will have the opportunity to write one from scratch! I did not know before that day that writing a compiler is something that people really do in practice as we live in a time, in which there already exist all these big, complex and seemingly perfect languages we use on a daily basis. Furthermore, it was revealed that the exam would not be a conventional written or oral exam, but a portfolio exam, in which for a set period of time we would be tasked with building different aspects and features of a compiler. It is safe to say, this was the perfect match for my ambition of learning more about compilers.

Doing the corresponding exercises for the course was challenging and established important recurring design patterns like visitors, matchers, and more. The time and effort investment in each sheet helped a lot for when starting to program the portfolio project, as the structure was familiar and mostly required concentrating on the "what" and "how", not the "where". While the freedom that you are given for implementing a specific task was quite welcome, there also came a negative with it. If one programs in a direction, which in time turns out to be wrong, he would need to almost start over, costing a lot of time. Because the exercises were not obligatory, one could still get away with not doing everything perfectly. However, what for me was most valuable is the experience I accumulated while doing the exercise sheets.

I began programming the portfolio on the first day the task description was released. My original idea was to start with making all the required changes in the grammar, but after having some troubles in understanding the supposed behavior of our language's multidimensional arrays, regarding the empty brackets having no specified length, I postponed the multidimensional array implementation until later. The main reason was that the modifications in the parsing phase do not only have to do with changing the language that is accepted by our compiler, but also with creating the corresponding data structures to convey its meaning. Therefore, I just needed a bit of time to process how the internal representation of arrays in our compiler needs to be changed to reflect the intuition of multidimensionality.

The parsing and analysis phase for interfaces and classes was quite straightforward and was done by the third day the exam had started. It felt really good writing a large amount of code for a relatively small amount of time. Since the template's grammar included all necessary

structures for the parsing of inheritance of classes, I had the impression that inheritance is also part of the exam's requirements. By the time I understood that this is not the case, I had already implemented its analysis. However, it did not cost me a significant amount of time and I do not regret it. Thinking back, I would have probably done it even if I had known that it is not required to do so.

While programming the analysis phase for interfaces, I was confronted with my first challenge. As at the time of parsing, our compiler does not know whether a type identifier is an interface or a class, it was necessary to create a new *NQJType* called *NQJTypeInterfaceOrClass* in the *.ast* file in order to convey this meaning to the analysis phase. The analysis phase can then find out what it is, having access to all interface- and class declarations. Before I came up with this simple and quite obvious solution while running my favorite route on a chilly morning, I tried various other wrong modeling ideas. This experience taught me that it is never a good idea to start a new, big addition to your program too late at night. As the russians say, "the morning is wiser than the evening".

After the analysis of interfaces and classes, it was time to work on their translation. This took a bit longer and involved trying out many ideas before a clear picture of the translator's workflow and behavior was formed in my mind. At first, I still wanted to translate class inheritance, but I quickly decided against implementing it, as it would have just complicated the task unnecessarily. Additionally, it was my intention to implement both class methods and class fields at the same time. Even then, it seemed to not be a brilliant idea, but my assumption was that in the case that everything goes wrong, making me start over, I could still reuse the written code as a basis for later versions. As expected, nothing in the translation of classes worked. After a few days of commenting out code and trying to debug each aspect of the translation separately, I decided to approach the task in a more modular way, as Prof. Bieniusa had also advised us to do before the start of the portfolio exam. First, my goal was to implement class fields, which was done fairly quickly and without any issues. Then, it was time for methods, which for the most part was a game of finding out how to provide the data needed in the many parts of the translation phase. Usually, I started with the bottom-most instruction of the *minillvm* wrapper library in a translation operation and worked my way up. This was immensely useful, since it enabled me to understand what kind of data I needed in order for a translation to work. After that, I mostly had to create new data structures (usually maps), fill them and use their content where necessary. Throughout the portfolio project instead of the usual "get" methods of data structures, I tried to consistently take a caching approach, which would return the element if present in the data structure and if not, would create it (or translate it) first and the return it. This was done as a "fail-safe" mechanism, making sure that for example constructor procedures are always translated, no matter when.

The hardest part of the translation phase was undoubtedly working with pointers. This was especially the case while trying to implement method calls. It was often very difficult to keep in mind what the object that you are working with really is. Is it a pointer to an object? Is it a pointer to a pointer to an object? This is obviously difficult to debug. Talking with other participants of the exam did not really help at that point, since many of them were still concentrated on the analysis phase. However, what helped me solve these problems was making a visualization of the data-flow on paper.

The idea of translating interface objects is in a way quite straightforward. Since underneath every interface object there must be a class object, which implements the interface, there are in theory almost no adjustments that are needed, provided class translation is implemented. This is why I decided to save in the analysis phase for each assignment of the pattern "*Interface1* **a** = **new** *ClassA()*", where *ClassA* implements *Interface1*, the class' reference in the internal interface object. This way, I was able to match each interface variable to its class object in the translation phase. However, much later while writing more tests it turned out that there is are two scenarios, which do not work as expected, regarding interfaces as argument types of functions and interface polymorphism. Hence, if there is one thing that I would change if I had to start over, it would be the handling of interfaces in the translation phase. Possibly, I would not set class objects to interface representations in the analysis phase, but actually the other way around. This way, each class object will be able to know what interfaces it really is. However, I do not think that the general idea is entirely wrong, but only that the timing where we assign a class object to an interface object is not correct. It would make more sense to handle this in the translation phase: whenever a new assignment to a temporary variable, containing an interface variable is made, we update its type accordingly too. This is currently missing in the project. Due to the time constraints, I decided not to spend much further time looking into this issue. This would be the first thing to be changed for a second version or iteration of the project.

After implementing each small part of the portfolio I was testing its correctness by writing small file unit tests. There were numerous times were the test file was not a valid *notquitejava* program and instead of proof-reading the code, I immediately questioned my implementation. In most cases, it turned out that I was using some syntax that is not correct, for example a declaration and assignment as a one-liner: "*int* **x** = **2**;". Therefore, it is safe to say that my compiler knows its language much better than I do, and I should probably be questioning myself more often, rather than directly blaming computers for my own mistakes.

All in all, I was done with the parsing, analysis and translation of classes and interfaces in less than two weeks. After that, I took upon the task of implementing multidimensional arrays. The first challenge was finding out what kind of arrays are accepted by our template. This was a bit confusing at first, since the original accepted arrays exhibited some characteristics of multidimensionality. In particular, the compiler accepted nested arrays, i.e. multidimensional arrays, in which each component is of length 1. Once I had a clear picture of the status quo, I was quick to make the necessary modifications on the grammar level and in the analysis phase. The translation, however, was not as simple. As I was struggling to understand how the template's implementation of one-dimensional arrays works, I was hesitant to try things out. I felt like there are just too many question marks in my mind, regarding what an implementation of multidimensional arrays should look like, and so I did not know where to even begin. I tried some ideas out, but I could not be sure that I am working in a correct direction. Since I did not manage to do the whole implementation, I have no idea whether I was close or far to a working translation of multidimensional arrays. Additionally, outside factors like for example other university exams were taking priority over spending even more time implementing this smaller feature in the grand scheme of the portfolio.

If I had to choose one aspect of the portfolio that I am most proud of, it would for sure be the

translation of classes. In the beginning I was so frightened of the whole topic of V-Tables, of all these data structures that I need and of all the intimidating *minillvm* wrapper functions, that I could not imagine how I would implement such a complex translation. This is also the aspect of the project, which was hardest to implement, but also the one, which I learned from the most. Even though I did not manage to implement multidimensional arrays, I stand by my view that the translation of classes is much more difficult to do right in comparison to the translation of multidimensional arrays.

In conclusion, I would shortly like to share my thoughts on the topics that were presented in the lecture. Each class, I learned so many new things about compilers and programming languages as a whole that my way of thinking about programming changed too. Mostly in regard to doing code optimizations by hand. Every one of the in the lecture presented topics served a purpose and taught me quite much. What I was personally looking forward to seeing is more discussions on the topic of garbage collection, which, understandably, due to time constraints was cut a bit short. In particular, I would be interested to understand how some programming languages approach this problem by adapting their type system (e.g. *Rust* [3]). . That being said, the basic algorithms and ideas in relation to garbage collection were presented, and one could probably do a full lecture just on that topic alone.

Lastly, I would like to thank everyone involved in the course. Special thanks to Prof. Dr. Annette Bieniusa, Mr. Albert Schimpf and my group partner for the exercises Mr. Gabriel Sánchez Gänsinger for making this lecture so interesting and fun.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] Marian Hristov. Repository. `https://softech-git.cs.uni-kl.de/students/clp/ws22/portfolio/hristov`, 2023.

[3] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.