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

Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Portfolio Exam

January 25 - March 8, 2023

Gabriel Sánchez Gänsinger

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

1 Parser

1.1 Changes to ast

The Ast file of our project was mainly modified to include Classes needed to represent parts of our extended grammar. First, to represent Interfaces we added *InterfaceDecl*. Similar to *ClassDecl*, an *InterfaceDecl* is created when an interface is declared. This class has only two attributes, a string name and a *InterfaceFunctionDeclList*, as our interfaces only define methods. This *InterfaceFunctionDeclList* is composed of one or more *InterfaceFunctionDecl* which are similar to the *FunctionDecl* already defined in our template. Similar to class method declarations, interface method declarations have Attributes: Type returntype, String name, VarDeclList formalParameters, but lack the method body, as we cannot implement methods in our Interfaces.

Furthermore, to make it able for classes to implement Interfaces we declared the class **Implements** which can either be *ImplementsNothing* or a *ImplementsInterfaceList* which consists of one or more *ImplementsInterface*. These *ImplementsInterface* only have a single attribute a String with the name of the implemented interface. ClassDecl was modified by adding a field Implements impl.

Some minor modifications made to the AST file, were the expansion of *NewArray* with a field Expr arraySize, to accommodate multidimensional arrays. The definitions of all the lists used in the implementation of Interfaces were also added.

1.2 Changes to cup

The grammar was extended as described in the portfolio specification. First, the description of the multidimensional array construct was introduced. For this, we created a new Non-terminal called *exprBrackets* which, similarly to *brackets*, accepts a succession of brackets, filled with expressions. These expressions are appended to a list to be used later. Instead of the definition of arrays given in the template, where only one size was defined, we insert the Non-terminal *exprBrackets* as seen in Listing 1.2. To pass the arrays we use the same helper method, that was used in the template to store one-dimensional arrays. The modifications to these helper functions are described in subsection 1.3.

```
brackets ::=
    LRBRACKET brackets:b
        {: RESULT = b+1; :}
  |
        {: RESULT = 0; :}
  ;

exprBrackets ::=
    LBRACKET expr:e RBRACKET exprBrackets:b
        {: RESULT = b; b.addFront(e); :}
    |
        {: RESULT = ExprList(); :}
;

expr2 ::=
    NEW baseType:t exprBrackets:size brackets:b
        {: RESULT = AstHelper.newArray(t, size, b); :}
  | NEW ID:t LBRACKET expr:size RBRACKET brackets:b
        {:  NQJExprList l = ExprList();
        l.add(size);
        RESULT = AstHelper.newArray(TypeClass(t),l , b); :}
;
```

Interface declarations were also added to the grammar as specified. These *interfaceDecl* are, similar to *functionDecl* and *classDecl* also *topLevelDecl* of which a program has a list. The *interfaceDecl* themselves are defined as shown bellow in Listing 1.2 and for each interface declaration an InterfaceDecl object is created. An *interfaceDecl* can have a list of *interfaceMemberDecl*, which are the methods declared in that interface, these are passed on in an InterfaceFunctionDeclList.

```
interfaceDecl::=
    INTERFACE ID:name LBRACE interfaceMemberDeclList:members RBRACE
        {: RESULT = InterfaceDecl(name,members);:}
    ;

interfaceMemberDeclList::=
     interfaceMemberDecl:d interfaceMemberDeclList:l
        {: RESULT = l; l.add(0,d);:}
    |
        {: RESULT = InterfaceFunctionDeclList();:}
    ;

interfaceMemberDecl ::=
    type:t ID:name LPAREN paramList:param RPAREN SEMI
        {: RESULT = InterfaceFunctionDecl(t,name,param);:}
    ;
```

Finally, for the implements keyword in a class, the Non-terminal *implementsList* was added, which accepts a comma-separated list of IDs that are inserted into an ImplementsInterfaceList. A class definition can have exactly one of these lists. If none is declared, we generate an ImplementsNothing instead

```
implementsList::=
  ID:i COMMA implementsList:l
    {: NQJImplementsInterface inter = ImplementsInterface(i);
    RESULT = l; l.add(0,inter);:}
| ID:i
    {:NQJImplementsInterface inter = ImplementsInterface(i);
    RESULT = ImplementsInterfaceList(inter);:}

;
```

1.3 Changes to AstHelper

The last change we want to mention in the parser phase is the modification of the helper function *newArray(NQJType **t**, NQJExprList **size**, int **dimensions**)* in AstHelper.java. This function defines the type of the NQJNewArray object that is passed to the analysis phase. In the template, the type of the array was composed of a nested type consisting of a base type nested inside of as many NQJTypeArray objects as there were dimensions. In our case, we added a field to NQJTypeArray to express the sizes of the defined dimensions. In this way, we nested the type, exactly like the given approach, but added the expression defining the size to each level.

```
public static NQJExpr newArray(NQJType t, NQJExprList size, int dimensions) {
    for (int i = 0; i < dimensions; i++) {
        t = NQJ.TypeArray(t, NQJ.ExprNull());
    }
    for (int i = 1; i < size.size(); i++) {
        t = NQJ.TypeArray(t, size.get(i).copy());
    }
    return NQJ.NewArray(t, size.get(0).copy());
}
```

2 Analysis

2.1 Nametable.java

The class Nametable defines an object that is used to store important information for the Analysis phase. This object is created at the beginning of the analysis and receives the analysis object and the *NQJProgram* as its parameters. In the constructor, the *Nametable* takes the *NQJProgram* and looks at its *TopLevelDeclarations*.

For each interface declaration, the *nametable* creates new *InterfaceType* objects with their name and the defined functions. These *InterfaceTypes* are then stored in an internal Map named ***interfaceDeclarations*** with their name as the key. For each class declaration, the *nametable* works similarly to interfaceDecls and creates *ClassType* objects with the relevant information and stores them in a similar map named ***classDeclarations***. Furthermore, the *nametable* checks for each interface, that the class implements, if the *classDecl* defines the same functions as the *interfaceDecl*. If not it adds an error to the analysis object. Besides this method, nametable has getter methods to get Class- and InterfaceTypes.

```
for (NQJClassDecl c : prog.getClassDecls()) {
        ClassType t = new ClassType(analysis, c.getName(),
                c.getImpl(), c.getMethods(), c.getFields());
        if (c.getImpl() instanceof NQJImplementsInterfaceList) {
            NQJImplementsInterfaceList list = (NQJImplementsInterfaceList) c.getImpl();
            for (NQJImplementsInterface im : list) {
                InterfaceType interType = interfaceDeclarations.get(im.getName());
                for (NQJInterfaceFunctionDecl f : interType.getFuncMap().values()) {
                    if (t.getFunc(f.getName()) == null) {
                        analysis.addError(c, "Has␣not␣implemented␣function␣"
                        + f.getName());
                    }
                }
            }
        }

        classDeclarations.put(c.getName(), t);
    }
```

2.2 ClassType.java

ClassType is an extension of the class *Type*, that handles the typing and subtyping of classes. The constructor has five parameters: the analysis object, the name of the class, the *NQJImplements* object from the parser phase, a list with function declarations, and a list with variable declarations. The name of the class and the *Implements* are simply stored in the fields of the class. The fields and methods of the class are stored in maps with the name of the field or method being the key. Apart from that, the constructor checks if there are not two methods with the same name and adds an error to the analysis object if this is the case.

The most important method of any class that extends our Type class is the *isSubtypeOf(Type **other**)* method. This method checks if a given *Type* is a subtype of another that is given in the arguments. In the case of class the method first checks if the other Type is also a *ClassType*, and returns a check on the equality of the names. If the other *ClassType* has the same name as the object they are the same Type, ass classes in our model have unique names. If ***other*** instead is an instance of an *InterfaceType*, the method checks if the name of the interface is inside of the *ImplementsInterfaceList* that was set by the constructor. Here again, we check on the equality of the name, as Interfaces also need to have unique names. Lastly, if ***other*** is neither of type *ClassType* or *InterfaceType*, the method returns a check on *Type.**ANY***, as this is the universal type and every other is a subtype of ***ANY***.

2.3 InterfaceType.java

Similar to *ClassType*, *InterfaceType's* constructor sets the name of the interface and creates a map with the defined functions, again checking on duplicates. The *isSubtypeOf(Type **other**)* method in *InterfaceType* is easier than the one in ClassType, as an interface in our model can only be a subtype of itself and ***ANY***. Therefore we check the equality of the name if other is an instance of *InterfaceType* and returns ***other*** == *Type.**ANY*** if it is not.

2.4 Analysis.java

This class is the main class of the analysis phase. It extends our *DefaultVisitor*, which implements visit methods for all nodes in our *NQJProgram*. Apart from that the analysis class implements the *check()* method which is the main method for the analysis phase. This method first creates a new *NameTable* object, then verifies the main method, which was already implemented in the template, and then starts visiting the *NQJProgram*.

We needed to adjust four visit methods in our Analysis class namely the methods for the nodes: *NQJClassDecl, NQJInterfaceDecl, NQJInterfaceFunctionDecl, NQJStmtAssign.* In the class declaration, we get the *ClassType* for the class from the nametable and define a new *TypeContext* with it. This context is then filled with the fields of the class, before pushing it to the ctxt field of Analysis. After this, the method invokes the visit method for all the *NQJFunctionDecls* of the class, which then can use the pushed context to check the method.

For *interfaceDecls* we do not need to check the fields of the interface, as they do not have fields. Therefore the visit method for *NQJInterfaceDecl* is limited to calling the visit method of its *NQJInterfaceFunctionDecl.* The visit method for *NQJInterfaceFunctionDecl* only checks that all parameters of the method are unique.

Lastly, a small modification in the visit method for NQJStmtAssign was made. This modification is meant to help in the later stages of the analysis and translation phases, as it sets a value for the Initialized field in *InterfaceType* if a variable that is of type *InterfaceType* is set to a *ClassType.* This field is then set to the corresponding *ClassType*, to be able to check other aspects later on. This modification is a needed one but it may cause errors when assigning a variable several times.

Apart from the modifications to *visit()* methods the *type(NQJType **type**)* method in *Analysis* was also modified, extending it with the *case_TypeClass()*, which gets and returns either a *ClassType* from the nametable, or an *InterfaceType* if no *ClassType* was found.

2.5 ExprChecker.java

This class, as its name suggests, type-checks all expressions. It extends matchers for *NQJExpr* and *NQJExprL* and has cases for all those nodes. In our project, we needed to implement the cases for the expressions relating to classes, and extend those relating to arrays to suit multidimensional arrays.

For classes, the first node that is going to be visited is the *NQJNewObject* node, which represents the expression new B() for class B. In our implementation, we simply check the timetable for a *ClassType* matching the name of the expression and check if it returns a non-NULL value. If this is not the case we add an error to our analysis object and return *TYPE.**ANY***. If the nametable returns a correct *ClassType*, we return this as our nodes return statement.

Once we created a new object of a class we can access its fields and methods. This is handled by the nodes *NQJMethodCall* and *NQJFieldAccess.* As these two methods are similar we will look at them together. In both cases, we get the name of the receiver and execute *check(NQJExpr **e**)* on it. This will return the type of the receiver as it was declared, so it either would be a *ClassType* or an *InterfaceType.* This is a problem, as if it is an *InterfaceType*, we would not know whether the variable got assigned an object of a class nor what class it got assigned to. This is where the value Initialized of *InterfaceType* plays a role. This is set whenever a new object is assigned to a variable that was first declared as an interface. In these methods, we get the *ClassType* that was assigned to the *InterfaceType* to execute further type checking. This again may lead to errors, which we will look at in **??**. After getting the correct *ClassType* both methods check if a method/field with that name exists in that class, and in the case of the method call, we also check if the parameters of the call are correct. This is done by checking the number of parameters and comparing the type to the expected one.

The other change that was made to this file, was to the node that is called when an *NQJNewArray* is created. The implementation given in the template checked the expression defining the size of the array and created a nested *ArrayType*, which was then passed. In our implementation, we go over the nested *TypeArray* of the parser phase, created in AstHelper, and check all expressions defining the size of the array. This is done in the method *checkBaseType(NQJNewArray* ***newArr****, NQJType* ***basetype****, ArrayType* ***arrType****).* If an expression is found that is not of type INT or NULL, as those are the types we expect, we add an error to the analysis object.

3 Translation

3.1 Maps of Translator

The main class of the translation phase is Translator. It has the function *translate()* which is the function called to translate the code, and stores all relevant information. This information is stored in the form of maps to be easily accessed whenever it is needed. In this section, we will take a look at these maps. The maps that we implemented are:

- *Map<NQJFunctionDecl, Proc>* ***methodImpl***: Stores the method procedures to be used and called. It is indexed by the NQJFunctionDecl of the method which is unique for every method.
- *Map<String, TypeStruct>* ***classes***: Stores the class TypeStruct. this map is indexed by the name of the class, as these should be unique and is filled first with dummy TypeStructs, and then extended to have the correct structure, once a class is processed.
- *Map<String, Proc>* ***constructors***: Stores the constructor procedures of each class. This map is also indexed by the name of the class, as these need to be unique.
- *Map<String, Global>* ***vtables***: Stores the vtable Globals for each class, indexed by the name.
- *Map<String, TypeStruct>* ***vtableStructs***: Stores the vtable Structs for each class. This is necessary as we need the information of the Struct, which is not accessible from the Globals.

3.2 Translator.java

The translation of classes starts with the method *translateClasses().* This method is called at the beginning of the translation phase in the method *translate().* It consists of several for loops that iterate through the classes and their methods. First, it goes through all class declarations of the *NQJProgram* and constructs dummy *TypeStructs* for each of the classes. This is needed, as later steps in the translation require to reference to a class *TypeStruct* before the class is processed. This is the case, for example, if a class has another class as one of its fields. If the parent class is processed before the child class the translation will not find a matching *TypeStruct* and fail. These dummies are put into the ***classes*** map.

The same is done to the methods of functions, as we need Procedures to be referenced before we translate the methods. The loop iterates through all methods and calls the method *initMethod(NQJFunctionDecl **func**, NQJClassDecl **class**)*. This method creates procedures for the function with all its parameters and adds a reference to the class *TypeStruct* as the first parameter. This first parameter will later be the ***this*** keyword inside the method. The result of the *initMethod()* call is then put inside the ***methodImpl*** map.

After both the classes and methods are initialized, the classes are translated. This is done via the call of the method *translateClass(NQJClassDecl **classDecl**)* on each of the class declarations in the NQJProgram. The first thing the method does is to call *initVTable(NQJClassDecl **classDecl**)*.

This method initializes the virtual method table for that specific class with as many procedure pointers as there are methods. The vtable is translated using the *Global* class, which gets the vtable *TypeStruct* and initial values for the procedure references and translates this to an llvm struct. *initVTable()* then returns the vtable *TypeStruct* to be used in *translateClass()*.

The dummy *TypeStructs* for the classes get then filled with *StructFields*, corresponding to the fields of the class declaration. The first field slot is reserved for the vtable. A *Proc* is created to serve as the constructor for the class. This *Proc* first allocates enough space for the struct on the Heap. Then it assigns a pointer to the vtable to the first slot of the allocated memory and fills the rest of the Fields with the default values for each type. After that, the procedure returns a pointer to the allocated memory. This procedure is stored internally in the ***constructors*** map, for it to be referenced later.

Finally, *translateClasses()* calls*translateMethod(NQJFunctionDecl **m**)* to translate the methods of the classes. The way that it proceeds doing this is exactly the same as the already implemented translation of functions, but skipping the first parameter, as this is the reference to the ***this*** keyword.

3.3 ExprRValue.java

The class *ExprRValue* handles the translation of all right-hand expressions. The expressions that are relevant for classes are *newObject* and *methodCall*.

When a new object is created we need to call the constructor for the corresponding class. This is what we do in *case_NewObject(NQJNewObject **e**)*. We get the constructor from the ***constructors*** map of Translator, define a new temporary variable and add a call to the constructor, which saves the result to the temporary variable. Finally, we return a reference to that temporary variable.

Method calls are handled in the method *case_MethodCall(NQJMethodCall **e**)*. First, we get the receiver of the method call by calling *exprRvalue(NQJExpr **e**)* on *e.getReceiver()*. This returns us the corresponding operand for the given expression. In our case, this is the variable on which the method is called on. From this variable, we can calculate the TypeStruct of the class with the methods *calculateType()* and *getTo()* which return a *TypePointer* and the struct that is in that given pointer. With the *TypeStruct* we find the index of the method in the corresponding VTable. Then we add instructions for getting the pointer of the vtable, loading the vtable, getting the pointer of the specific procedure, and then loading it. All these are stored in temporary variables. After getting the corresponding procedure, we get the parameters from the method call, cast them to the corresponding type if necessary, and add them to an *OperandList*. The first parameter is the receiver itself, which in our case is the ***this*** parameter.

To conclude we add a call to the procedure with the parameters we got, and return the resulting temporary variable as a reference.

3.4 ExprLValue.java

The class *ExprLValue*, similar to *ExprRValue* handles the translation of all left-hand expressions. In our portfolio, we needed to modify the behavior for two of these expressions, *fieldAccess* and *varUse*.

FieldAccess starts similarly to methodCall, as it needs to get the receiver of the field access using *exprRvalue(), calculateType()* and *getTo()*. Once it has the corresponding TypeStruct, it searches for the index of the desired field, to then adds an instruction to get the pointer of the field and return it.

The modifications made to *VarUse* were to be able to use the fields of a class, when inside a method call. If the variable is not found in the ***LocalVarLocation***, which is a map in translator, that stores all locally available variables, we need to check if it is a field. We do this by getting the ***this*** parameter and getting the class TypeStruct from it, as done before in methodCall and fieldAccess. Similar to fieldAccess we then look for the index of the Field and add an instruction to get the pointer for that field.

4 Tests

4.1 Analysis tests

The tests for the Analysis phase are separated into the file Portfolio_test_analysis.java and several files for *FileAnalysisTest.java*. The tests *testSimpleClassWithMethod()* and *testSimpleClassWithField()* are as their names suggest, simple tests to check if the main functionality of the classes is being correctly type checked. In both a class is created and either a method is called or the value of a field is modified. In *testClassandInterfaceInteraction()* we check if the interaction between interfaces and the classes that they implement work correctly and if we check that a function corresponds to that interface/class. In *testClassFieldAccess()* we check if the analysis phase can handle field access from outside and inside the class. In *testInterfaceCall()* we try passing an interface to a function and see if methods can be called.

The files for the File analysis tests are located in *testdata/typechecker/error/Portfolio* and in *testdata/typechecker/ok/Portfolio*. All added tests are prefaced with the word *Portfolio_*. In *ArrayCreation.java* we check if an error is thrown if we try to initialize an array with no determined length in any dimension. In *classNoImplement.java* and *classNoMethod.java* we check if errors are thrown if the implementation of an Interface is not correct, so for example if the class has no keyword implements and is assigned to an interface and if a class does not implement a method that the interface has declared. The tests *interfaceNoField.java* and *interfaceCorrect.java* are designed to be opposite of each other, as the first should throw an error as we cannot call fields on interfaces, and the second demonstrates how it should be done correctly. Finally, the test *MultipleInterfaces.java* checks if the type-checker can handle the implementation of several interfaces by one class.

4.2 Translation tests

The tests of the translation phase are similarly separated into a file Portfolio_tests_translation.java and some file translation tests. In the first file, we have *testClassAssignment()* which tests the assignment of default values to the fields of classes and the modification of the fields. *testClass-Method()* tests the interaction between methods and fields inside a class, while *testInterfaceCall-RunsThrough()* is the same test as *testInterfaceCall()* but in the translation phase. For arrays, we have the test *testArrayAccess()*, which tests multidimensional array creation and access.

The files for the file translation tests are similarly located in *testdata/translation/Portfolio*. *Portfolio_MethodWithInterface.java* tests the interaction between different classes that implement an interface and a function that expects an argument of that interface. Lastly, the test *Portfolio_polymorphism.java* was made to display errors in the handling of interfaces.

Reflection

The portfolio exam was an interesting project where I learned a lot of things about what tasks I can perform correctly and efficiently and which ones I need to further work on. I took the opportunity to work on some of the aspects and try to improve them, but at last, I think that I still need a lot of practice in some aspects.

But let's start at the beginning, the initial assignment of the portfolio. As it still was in the middle of the semester, other lectures got in the way and I couldn't get much done. I started thinking about my approach to solving the task and implemented the parser phase pretty quickly. As I noticed how quickly I had implemented this phase, my expectations of the work needed were lowered, as I thought that it would all go the same way. This was my first mistake, underestimating the work needed. I had planned to implement most of the bulk of the program in the first three weeks, as it would give me time for the documentation and reflection, which I thought would be the most problematic task, given that I usually am not good at writing long texts. This was an unrealistic goal, as soon I would notice that between university and family, I would lose at least one week if not more. And given that the implementation of the later phases was harder than I expected, this shifted my timetable quite a bit. So much so that I still am not finished with the things that I wanted to implement and need to hand in the project with some key features either missing or not working quite correctly.

This is probably a good time as any other to mention what I have done in this project. I implemented the whole translation of classes with fields and methods. Which in the end works properly and I am proud of it. The ideas behind these implementations came from discussions with other students or from myself, but the whole implementation and problems that arouse were a fun bit to work on and to figure out by myself. In terms of the interface implementation, this was also implemented. There are some problems that we will discuss later, but as far as working interfaces go, the requirements that were set in the portfolio were met, at least in my opinion. Then we come to multidimensional arrays and the lack of understanding of them. I implemented the parsing and analysis of multidimensional arrays quite quickly. I thought that I had done a good job and was eager to try to implement them in the translation phase. But when it came to that I had problems in regards to how the backbone of our compiler worked in terms of arrays. The problem was not with the llvm structure of arrays and how these should look, but with the information passed by the analysis phase and how we should use that information. This was also at the time that I started worrying about time and an exam that is nearing and will be hard to study for and try to pass. All these things, added to some personal experiences made my mental block in terms of implementing the translation of arrays somewhat difficult. I now know how I would maybe go ahead to try to implement them if I had more time. But the project as a whole would require an entire remodeling in my opinion and I would not have the time to do it right now.

This brings me to my second mistake, modeling the interfaces and their relation to classes. Quite early in the project, during lunch with a fellow student, we came to the idea of modeling, which would make it easy to implement interfaces and the correlation to the classes implementing them. The main idea was to assign class types to the interface types when they got assigned. In this way, we would be able to check if a variable was initialized, and would be able to easily access the information that was relevant in the translation phase. In this way, an interface was bound o a class implementation and we would not need to worry about interfaces in the translation. This is wrong, as we initialize one interface type for each interface declaration, which then gets assigned exactly one class. This is not a problem if the classes that are assigned have the same methods, aka. they only implement methods that are inherited from that interface. But as soon as the classes implement other methods and with that alter the order of methods inside the class struct, we can see the problem with this implementation. If we look at an example we could have a class assigned to a variable defined as an interface. This class has 1 method which is inherited from the interface. This class' class type is assigned as the initialized value of the interface type. If we then override the assignment of that variable with an object of another class, that instead has 2 methods, of which the first one is different from the other class, the translation goes over the initialized class in the interface, gets the index of the method, which would be 1, and tries to call this method on the second object. As this class has another method in the first index, the program outputs something incorrect or fails. This problem is hard to fix once in the translation phase, as it depends hardly on the type of construction given in the template, which we expanded. Furthermore, as llvm needs fixed types, we cannot simply override certain parts of the implementations where types are needed and work with simple pointers to try to avoid the problems. The fix would require a total remodeling of the analysis phase and the data structures that we pass to the translation phase. But as this problem was spotted rather late into the project, I had not enough time to do such a radical change to the program.

This leads me to my final mistake, my approach to how to tackle the project. In other subjects such as Software engineering 3, Foundations of software engineering, and others we are told the importance of requirements engineering and to get a good idea of the project before we start. Implementing black box tests beforehand and working with them to achieve the requirements of the project. It seems that all those tricks and recommendations were not in my head at the beginning of this task. I had an idea of how the project would need to be and how it should be executed. This was only cemented as I saw others with similar ideas and concluded that I was doing things correctly. But as soon as I started to think about test cases and tested them to check my implementation I started to see the cracks in my thinking. If I would have started by examining the task at hand and looking at the requirements more deeply and defining test cases only with those, some problems with my way of thinking would have been exposed at the beginning and would have not been propagated throughout my entire implementation. If I would do this project again, which is entirely possible, as I find it a good exercise, I would start by planning a week of my allotted time, to write a somewhat informal functional specification document and write tests, based on that. This would help not only with the correctness of the final product but also with time management, as I would not have spent a long time trying to fix errors.

However, this project still made me learn a lot of things and practice others. I have never worked on a large project like this, apart from maybe the SEP project in my bachelor's, and it has given me perspective on how I work and what needs to be improved. For once, I like that I feel more comfortable working on projects that were already started. I understood the template relatively easily, apart from some kinks in array translation as already mentioned, and could start working on it. This is also due to having done the exercises, but I think that I can adapt easier to new environments. Debugging, on the other hand, is one thing with which I struggle, as I am not accustomed to using the tools given by the IDEs. In this project, I could train myself to use these tools and get more familiar with them. What did not help with this was that the environment that I had built for this project (Windows with a built-in Linux installation), was not the best for debugging. For the parser and analysis phases, I could use the debugging tools integrated with IntelliJ that work very well. But as soon as I had to debug something in the translation phase I could only debug parts of the code with IntelliJ, as I could not bring commands like lli or clang to work on my windows system. This made it only possible to run the generated code on the Ubuntu installation inside my windows ecosystem, which works very slowly. If I were to try this project in the future again, I would try to refurbish one of my old laptops and try to install a Linux distribution to make it easier.

In general, this class and exam have thought me a lot of things about not only the subject itself, but skills that I have and should hone, or skills that I need for my future work. The topic itself was very interesting and something that will help me going forward. I had an idea of how compilers work beforehand, but that got completely changed as we progressed through the course. The most interesting part is the translation of the language and how you can use intermediate languages to help you. For some reason, I always thought that compilers translated directly to the most basic Assembler code and struggled to see how you could do that translation. Therefore, seeing a complete walkthrough through all the steps, helped me clarify my questions. This was helped by having hands-on experience with a compiler and being able to implement one.

References

[1] Gabriel Sanchez Gänsinger. Project repository. https://softech-git.cs.uni-kl.de/students/clp/ws22/portfolio/g_sanchez17, 2023.