A modular compiler for MiniJava

Patrik Reppien
s103092

Kongens Lyngby 2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The purpose of this project has been to provide a motivating and exciting basis for learning about compilers, by enhancing an existing basis. The project consisted of two parts.

The goal of the first part was to analyse an existing MiniJava-compiler used for teaching purposes and use this analysis to provide a layer of persistence between each internal module of the compiler. Enhancing the compiler to be externally modular in this way makes the compiler more suited for educational purposes, as students would be able to implement a module and have the two missing modules handed out as binary files, resulting in a functional compiler. The challenge in this part of the project has been to provide a layer of persistence which does not require changes within the classes to be persisted, by requiring data to be exposed or annotations on fields. It was decided to implement a custom serializer and a matching deserializer from scratch for maintaining the layer of persistence. This proved to be difficult and many special cases had to be resolved including loss of references, private field, and circular references. The serializer and deserializer ended up being fully functional, and the layer of persistence was implemented successfully. This was successfully tested by using the original compiler to compile a number of test files, which were then compared with the result of compiling the same test files with the enhanced compiler.

The second part of the project concerned implementing a module for the enhanced compiler which was able to generate executable byte code for the new Lego Mindstorms EV3 robot. Documentation for the assembler language was found along with a working assembler which was successfully used to turn a hello world assembler program into a file running on the EV3. The new goal of the EV3 module now became to compile MiniJava into this assembler language, which would then be able to be assembled and executed on the EV3. This assembler language proved to be very different from the assembler language compiled to in the first part of the project. A memory managing system was implemented in the new module along with an Application Programming Interface, allowing the MiniJava language to interact with the special features of the EV3 such as the display. Most existing features of the original back end was also implemented in the new module.

Resume

Formålet med dette projekt har været at skabe en spændende og motiverende basis for at lære om compilere, ved at forbedre en eksisterende basis. Projektet har været opdelt i to dele.

Målet for den første del af projektet har været at analysere en eksisterende MiniJava-compiler brugt til undervisning og bruge denne analyse til at skabe et persistenslag mellem hvert internt modul i compileren. Ved at forbedre compileren til at være ekstern modular bliver compileren mere egnet til undervisningsformål, da brugeren vil være i stand til at implementere et modul og få de to manglende moduler udleveret som binære filer, hvilket vil resultere i en funktionel compiler. Udfordringen i denne del af projektet har været at skabe et persistenslag som ikke kræver at der bliver ændret i de klasser der skal persisteres ved at stille krav til udsatte felter eller anmærkninger på disse. Det blev derfor besluttet at implementere en serializer og en dertilhørende deserializer fra bunden af til at opretholde persistenslaget. Dette viste sig at være svært, og der skulle tages højde for mange specialtilfælde, her i blandt tab af objectreferencer, private felter og cirkulære referencer. Serializeren og deserializeren endte begge med at være fuldt funktionelle, og persistenslaget var derved oprettet. Dette blev gennemtestet ved at bruge den originale compiler til at compilere et antal testfiler, som efterfølgende blev sammenlignet med resultatet af at compilere de samme testfiler med den forbedrede compiler.

Den anden del af projektet omhandlede en udvidelse af compileren. Denne udvidelse skulle bestå i et modul til den nye compiler som var i stand til at generere eksekverbart bytecode til den nye Lego Mindstorms EV3 robot. Dokumentationen for det dertilhørende assemblersprog blev fundet, sammen med en assembler som med succes blev brugt til at oversætte et hello-world program til en fil som kunne køre på EV3 robotten. Det nye mål for EV3 modulet blev nu at oversætte MiniJava til dette assemblersprog, som efterfølgende kan blive assemblet og eksekveret på EV3 robotten. Dette assemblersprog skulle vise sig at være meget anderledes, i forhold til det assemblersprog der blev compileret til i første del af projektet. Et hukommelsesstyringssystem blev implementeret i det nye modul sammen med en Application Programming Interface der gør MiniJava-sproget i stand til at interagere med de EV3-specifikke funktioner så som skærmen. De fleste af den oprindelige backends funktioner blev også implementeret i det nye modul.

Preface

This report is the result of a 15 ECTS point bachelor's thesis conducted at the department of Informatics and Mathematical Modelling at the Technical University of Denmark, DTU. The thesis is entitled A modular compiler for MiniJava and is the result of work conducted from September 2013 to February 2014. The thesis is the final requirement for acquiring the degree of Bachelor of Science in Engineering, and has been written under supervision of Associate Professor Christian Probst.

I would like to thank my supervisor for valuable guidance and good advice throughout the project.

Søborg, 17-February-2014

[illegible]

Patrik Reppien

Contents

Chapter 1

Introduction

Compilers form the foundation of programming as we know it today. A compiler provide a programmer with a very high level of abstraction and enables the programmer to focus on creating the actual program instead of being forced to perform basic tasks like managing memory. This decreases the development time of programs significantly. In addition, languages with an even higher level of abstraction are likely to be expected in the future.
For these reasons, every computer science graduate should know about the inner workings of compilers. This project aims to provide exciting and motivating tools for effectively learning about this subject.

The aim of the first part of this project is to enhance a compiler written in Java that transform the educational language MiniJava into assembler language for the LC3 processor. The implementation of this compiler is to be examined and ultimately divided into three externally modular applications. An user can then be provided with two of the applications as binary .jar files and a program skeleton of the third. If the missing application is implemented correctly, the user is able to pass test programs through the whole compiling pipeline even though only one part has been implemented. This should motivate the user and increase the overall understanding of the compiler.

The second part of the project is an attempt to create a module for the compiler which is able to transform MiniJava into executable byte code for the new Lego

Mindstorms EV3 robot. The robot was released on September 1, 2013, hence not much experimentation has been done on the computer brick yet. As the result of this module will be executable code, able to be run on the robot, the module should be a motivational and interesting addition to the educational compiler.

This report can be read by anyone with a basic understanding of object oriented programming, data structures, and computer science modelling.

Chapter 2

Part 1: Creating a modular compiler

2.1 Introduction

The aim of this part of the project is to analyse and enhance an existing compiler, by making it externally modular. This will be achieved by examining the existing implementation and determine, whether this needs to be altered in order for us to separate the modules. After this, an approach will be chosen for achieving external modularity. The solution will then designed and implemented. Afterwards, the implemented solution is tested to ensure that the compiler still behaves as expected. Finally, the solution and decisions chosen during this part are discussed.

2.2 Analysis

In this section the language of MiniJava will be examined and described. The compiler being enhanced in this project will also be examined, described, and analysed, making it easier to modify the compiler in both this part of the project and the next part. Furthermore, the problem of achieving external modularity for all modules of the compiler is analysed, which includes an analysis of the internal modularity of the compiler. The notation used in this section and the rest of the project is explained in Appendix A.

2.2.1 MiniJava

MiniJava is a programming language used for educational purposes within many areas of computer science, including programming, algorithms, semantics, and compilers. The language itself is a subset of Java and removes the most advanced features from Java such as parallel threading, inner classes, and library includes. There are various versions of MiniJava. Some of these include extensions to make it easier to learn how to program and some more simplified than others[1]. The Backus-Naur form (BNF) grammar of the version used in this project is seen in figure B.2 found in Appendix B. From this grammar it is seen that the used language is still object-oriented, but does not include all object-oriented features of Java, for example abstract classes, and interfaces. Other features omitted are for-loops, exceptions and exception handling, switch-statements, and several logical and arithmetic operators. The grammar is examined further in section 2.2.2.2

2.2.2 The original compiler

A compiler normally consists of three phases: Front end, analysis, and back end. Each of these phases have different responsibilities and the relation between them is seen in figure 2.1.

The front end takes an input program and runs it through the lexical analyser, which converts the source code into a stream of tokens. Next, the stream of tokens is parsed to form the Intermediate Representation (IR), a tree data structure, representing the program in a way that makes it easier to analyse and translate.
In the analyse phase various things happen. While the syntax might be correct, types may not match or variables might not have been initialized when being read. Checking for errors like these is usually done in the analyse-phase. Fur-

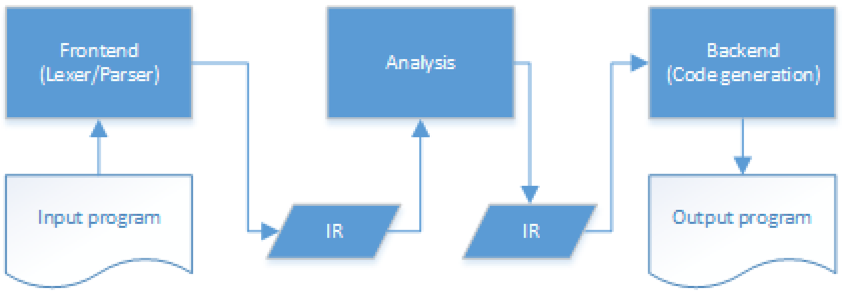


Figure 2.1: The three phases of a compiler.

thermore, the analyse-phase can be used to optimize and rewrite the code.
The back end, or code generator, translates the IR into the target language and usually also manages memory.
The compiler being extended in this project also consists of these phases and contains the following packets:

- `compiler` - The main compiler class and a pretty printer for debugging.
- `compiler.CODE` - Support classes for code generation.
- `compiler.CODE.LC3` - Instruction classes for code generation.
- `compiler.Exceptions` - Exceptions used in the compiler.
- `compiler.Frontend` - The front end containing the lexical analyser and parser.
- `compiler.IR` - Classes for the IR data structure.
- `compiler.IR.support` - Support classes used for analyzing the IR and for generating code.
- `compiler.Phases` - Classes used by the compiler to start the different phases.

The entry point of the program is the main method in the main compiler class. This class processes the command line arguments, which can be the following:

- `-v` for debugging mode.
- `-o` followed by a file name for defining the output file name.

- The filename of the source file to compile.
- −− for indicating that the rest of the arguments should go to the input program.

The file name for the input file is of course required. If this is not supplied, a help message is printed.
The compiler then executes the three phases sequentially while printing messages to the console, informing about the process so far.

2.2.2.1 The IR datastructure

Before the implementation of the phases mentioned above can be explained, the IR datastructure should first be examined. As the IR is the output of the front end and analysis, and the input for the analysis and back end, it is easier to understand these parts if one has an understanding of this datastructure.
The IR can be seen as a tree with the class `IR` as the root node. In this class, the main program of type `MJProgram` is contained in a field. In addition, some static fields with support objects are defined; a `MJClassTable`, making it easier to retrieve classes, methods, and fields, a `MJScopeStack` which keeps track of which variables that are able to be used, and two fields saving the current method and the current class.
`MJProgram` contains a list of the classes defined in the program. Each item in the list has type `MJClass`. Each `MJClass` contains an unique name, a list of methods of type `MJMethod`, a list of fields of type `MJVariable`, and the type of the superclass, which is of type `MJType`. Note that primitive types, like the name of the class, will be leaves in the tree-structure, as no further objects of the IR can be defined here.
The `MJType`-class is defined in order to help with type checking and code generation. In this class, an enum is used to define some standard types; the primitive int and boolean types, the `null`-type, and the type `void` for being used as a method return type. For treating constructors different from normal methods, a `constructor`-type is also defined. If a `MJClass` is passed as parameter to the `MJType`-constructor, a class type is created and the class used for creating the type is saved in a field that can be accessed with a getter. This means that it is possible to get the `MJClass` being the superclass of another `MJClass` by calling this getter on the field denoting the type of the superclass. The `MJType`-class is also used for array types. In that case an array type is created and the basetype-field will contain the type of the elements in the array.
The `MJMethod`-class also contains a name, a return type, and a list of parameters. Each entry in this list is of type `MJVariable`. The modifiers of the method are saved as three boolean values: isStatic, isPublic, and isEntry. A static method

means that the method can only operate on static fields of the containing class. If the method is public, it can normally be accessed outside of the containing class, whereas if it is private it can not. However, this has not been implemented yet, and this value is currently only used by the pretty printer, which prints the whole IR in a readable format. The isEntry is used to determine whether the method is the main method, and thereby the entry point of the program. Finally the body of the method is defined as a `MJBlock`.
In a `MJBlock`, a list of variables of type `MJVariable` and a list of statements of type `MJStatement` are defined. The latter type is an abstract class, and the statements can be one of the following:

- `MJComment`: A comment which consists of a single string. Comments are not compiled and only serve as a tool of documentation when coding.

- `MJAssign`: An assignment statement which consists of a left hand side and a right hand side. The left hand side is an identifier of type `MJIdentifier` which will be bound to a `MJVariable` at compile-time. The `MJIdentifier`-class is elaborated under the expressions later in this section. The right hand side is a `MJExpression`, stating the value that the bounded variable should be assigned.

- `MJIfElse`: An if-statement containing a `MJExpression` and a then-block of type `MJBlock`. The statement can also be supplied with an optional else-block. The statement will branch the program depending on the value that the `MJExpression`-evaluates to. If the expression is evaluated to true, the then-block is executed. If the expression is false, the else block is executed if present. If the else block is not present, the program continues.

- `MJWhile`: A while-loop which has an expression and a `MJBlock` as body. The semantics are similar to the if statement, but if the expression evaluates to true, the body is executed once, and the expression is evaluated once more. This is repeated until the expression evaluates to false or the program is aborted.

- `MJMethodCallStmt`: A method call without any return value. This can only be used as a statement, as no value for use in an expression is returned. The method contains an identifier for identifying the method to call and a list of `MJExpressions` which are used as parameters for the method. The method call statement is bound to a method at compile-time by looking up the method using the class table of the `IR`-class.

- `MJReturn`: A statement used for returning an expression from a method. The return statement is required as the last statement of a method except for the main method and can not be used anywhere else.

- `MJPrint`: A statement for printing the evaluation of an expression to the console. This statement would not be seen in a compiler normally, as this is covered by the method call statement explained above. However, as this compiler is for teaching purposes and does not include libraries, the `System.out.print` and `System.out.println`-statements are parsed into their own IR-class. This makes it easier to treat them differently from normal method calls when code is generated. It is possible to treat methods in different ways without having dedicated classes for them in the `IR`-datastructure, which is the approach that will be taken in the second part of the project.
- `MJPrintln`: A statement, where the only difference from the `MJPrint`-statement is that this statement will end with a new line character.
- `MJBlock`: A block for holding statements and variables. Blocks can be nested inside each other for creating a new scope for variables.

The abstract class `MJStatement` is used for defining operations that all statements should implement, such as pretty printing and code generation. In this way, polymorphism can be used to easily apply the same operations to different statements that implement this operation in different ways.
Most of the above statements use expressions for evaluating values. Like with statements, the `MJExpression` class is an abstract class used for defining common operations for expressions. A type is defined for each expression, which is useful for checking types when different statements and operators are applied to an expression. An MJExpression can be one of the following subclasses:

- `MJBinaryOp`: This is an abstract class that all binary operators inherit from. It defines two expressions denoting the left hand side and the right hand side of the operator, respectively. A subclass of this abstract class is either an arithmetic, a logical, a comparison, or a string operator. Below is an overview of all binary operators in the compiler:
  - Arithmetic operators
    - `MJPlus`: Used for adding integers.
    - `MJMinus`: Used for substracting integers.
    - `MJMult`: Used for multiplication of integers.
  - Logical operators
    - `MJAnd`: The logical $\wedge$ operator for boolean expressions.
  - Comparison operators
    - `MJEqual`: Determines if two expressions evaluate to the same value.

    * `MJLess`: Checks whether the left hand side of the expression evaluates to a value which is less than the value of the right hand side.

  - String operators

    * `MJPlus`: This operator is also used as a concatenate-operator for strings, appending the right hand string expression to the left hand string expression.

  Note that operator precedence is not handled by the data structure, but instead defined by the front end.

- `MJUnaryOp`: Similar to the `MJBinaryOp`, this is an abstract class that is inherited by all unary operators. The class contains a single expression that the operator is to be applied to. There are two unary operators in the compiler: `MJNegate` which represents the boolean negate-operator, and `MJUnaryMinus` which turns a negative integer expression into a positive one and a positive into a negative.

- `MJParentheses`: This expression class is used for giving some expressions higher precedence than others. The class contains a single expression which is the content of the parentheses.

- `MJIdentifier`: If an identifier is used in an expression, the value of the variable bound to the identifier is used. The `MJIdentifier` has two subclasses, `MJArray` and `MJSelector`. The first is an identifier for an array object which contains the identifier for the array and the index to be read from (in an expression) or written to (in an assign-statement). The second subclass is used for selecting fields and methods from an object in expressions or statements. The `MJSelector` therefore has two fields: An object-field determining what object is used to call the field or method and a field-field determining the field or method of the object that the `MJSelector` should be bound to. The two keywords "this" and "super" can be used to set the object field to the current class or to the superclass of the current class.

- `MJIfThenElseExpr`: This is the expression equivalent of the if-statement. The expression contains three expressions; a condition, a then-expression, and an else-expression. If the condition supplied evaluates to true, the then-expression is chosen. If evaluated to false, the else-expression is used. Note that the else-expression is not optional as in the if-statement, but has to be supplied.

- `MJNew`: An object initializer. This expression consists of a list of expressions denoting the arguments for the constructor. Through the declaration of the `MJType`-field of the `MJExpression`-superclass, it is possible to retrieve the MiniJava class that should be instantiated. The constructor of this class is then looked up and bound to the `MJNew`-expression. This binding is created in the analyse phase. The expression has the `MJNewArray`-class as a subclass, which contains the same information as `MJNew`, but as this expression has to initialize an array, a `MJExpression`-field is added which value is used to determine the size of the new array.

- `MJMethodCallExpr`: This is the expression equivalent of the `MJMethodCallStmt`-class. The semantics are the same as the statement-version of the class, except for the fact that the return value of the method will be the value of evaluating this expression.
- `MJCast`: Used for converting an identifier of one type into another. The type to convert into and the identifier to cast is saved in fields. The identifier is basically wrapped into the `MJCast`-expression, and the type of this expression then decides how the identifier is treated.
- `MJNoExpression`: This is basically a null value for expressions. Used for return statements in a method of type void and for variables that have not been initialized yet.
- `MJBoolean`: A boolean primitive, having either the value true or the value false. Both possible values are defined in the public and static True and False fields.
- `MJInteger`: An integer primitive.
- `MJNull`: Denotes the null reference for objects.
- `MJString`: A string literal, which contains a string value.

An overview of the class hierarchy of the IR is seen in Appendix C. For illustrative purposes, some associations and classes are omitted from the diagram and no associations with classes outside the IR are shown.

2.2.2.2 Front end

The front end is responsible for converting the text document containing the source code into the IR. As lexical analysers and parsers can be quite difficult to build and debug, several tools called *parser generators* exist to make the process of creating these easier. One of the most widespread parser generators generating java output code is the ANTLR-tool. This tool takes a context-free grammar as input and generates a Java parser that can read this grammar. The grammar is defined in a file with the file extension *.g*, and the file attachment

MiniJava.g contains the input used for generating the parser for the original compiler. The MiniJava grammar defined in figure B.2 in Appendix B is derived from this file. In addition to defining the grammar of the language that is to be parsed, ANTLR also allows the programmer to define what to do when hitting the different tokens. This is expressed by writing java code in the .g file at each tokenm and can be used to return objects and perform operations associated with this token. More specifically, this makes it easier to bind the parser to the IR by binding each token to a class defined in the IR hierarchy.
ANTLR also allows the user to define regular expressions. These can be used for defining fragments, which are alphabets for use in other regular expressions. The notation used for regular expressions is explained in Appendix A. In the MiniJava.g file, the following fragments are defined:

- `LOWER : ('a'..'z')` (Lower case letters)
- `UPPER : ('A'..'Z')` (Upper case letters)
- `NONNULL : ('1'..'9')` (Numbers from 1 to 9)
- `NUMBER : ('0' | NONNULL)` (Numbers from 0 to 9)
- `NEWLINE:'\r'? '\n'` (Optional carriage return and new line character)

In addition, a fragment CHAR is defined, which denotes all allowed string characters. From these fragments, regular expressions are defined for use in the grammar of MiniJava. These include the following:

```
IDENT : ( LOWER | UPPER ) ( LOWER | UPPER | NUMBER | '_' )*
INT : '0' | ( NONNULL NUMBER* )
STRING : '"' CHAR* '"'
COMMENT : ( '//' .* NEWLINE | '/*' .* '*/' )
WHITESPACE  :   ( ' ' | '\t' | NEWLINE )+
```

As the regular expressions imply, identifiers have to start with a letter and can then be followed by letters, numbers. and the '_' character. Integers are either a zero or a nonzero-number followed by additional numbers. A string is a sequence of characters enclosed in quotes and a comment is defined by two slashes followed by any text terminated by a new line or text between the /* and */ character sequences. Last, whitespaces are defined as any positive number of spaces, tab-character, and new lines.
The grammar in figure B.2 in Appendix B already describes how the programs should be structured in order for the front end to parse, and this grammar also declares where the different regular expressions defined above are used. The MiniJava.g-file will not be changed in this project, nor will the ANTLR-output parser. The MiniJava.g file or the ANLTR tool will therefore not be explained

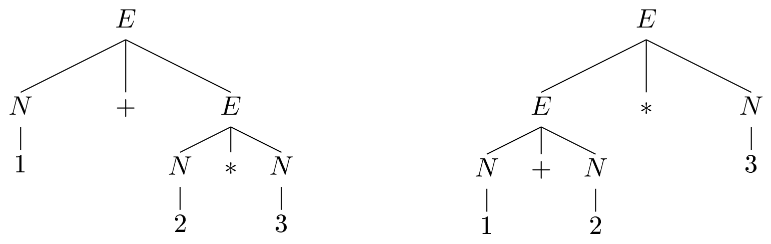


Figure 2.2: The expression $1 + 2 * 3$ parsed in two different ways, giving two different results when evaluated. E is an expression while I is an integer. The result of the left parse tree is 7 while the result of the right tree is 9

in great detail.[1]. However, most of the production rules of the grammar have a corresponding class in the IR. When a token matches a production rule, the corresponding class' constructor is called to create an object which is then returned by the production rule. In addition, types, modifiers, and other settings can be read and passed to the appropriate constructor or method. If a production contains the closure operator (*), implying that zero or more occurrences of a specific token is accepted, a linked list is used to hold the corresponding IR-objects.
Next, operator precedence is considered. The same expressions can have different parse trees representing different precedence hierarchies as seen in figure 2.2, where the *-operator has higher precedence than the +-operator in the tree to the left, while + has the highest precedence in the tree to the right.

It is important to make sure that all expressions are parsed the right way so they form the correct parse tree in order to obtain the operator precedence wanted. The front end of the original compiler handles this by defining six levels of parsing. Level one is parsed first, then level two and so on. The operators are grouped according to their level as seen below.

Level one: && (`MJAnd`)

Level two: == (`MJEqual`)

Level three: < (`MJLess`)

Level four: $+, -$ (`MJPlus` and `MJMinus`)

[1] More information can be found in the ANTLR documentation at: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation

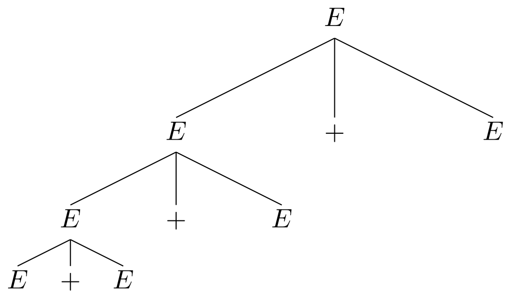


Figure 2.3: A left-associative parse tree.

Level five: $*$ (`MJMul`)

Level six: All unary operators, parentheses, primitives, and other expressions.

Note that these levels are defined in figure B.2 in Appendix B, where level one is named `expression`, level two is named `level1` and so on. This system ensures that the `&&`-operator has the lowest precedence and that all parentheses and unary operators have the highest precedence and hence are evaluated first when traversing the IR tree. All binary operators mentioned above are left associative, as the left expression is evaluated before the right. Expressions using more than one operator from a level will therefore form parse trees like the one seen in figure 2.3

2.2.2.3 Analysis

The analysis part of the compiler should ensure that the IR is ready for the back end, and that basic errors such as types not matching or uninitialized variables is caught before code is generated. The analysis phase in the original compiler consists of four sub-phases, which are explained below.

First, the IR is rewritten in the *first rewrite phase*. This phase finds all non-static methods of the program and add a parameter, *this*, with the same type as the class containing the method. This parameter is later used to pass the object containing the method as parameter to this method.

Next, the *type check phase* checks the entire IR for type errors. A type check method is implemented by most classes, and each of these methods ensure that the IR objects they contain are also type checked. This phase also checks

whether variables are actually in the scope when they are used.
Initially, the type check method of `MJProgram` is called. This method adds a MiniJava Object class and a MiniJava String class to the IR. Doing this makes the programmer able to use these classes without having to define them. A text and length-attribute is also added to the String class, although these are not assigned anywhere in the compiler, hence the programmer would have to assign these manually.
Afterwards the support classes are being set up; all classes are added to the class table, and all methods added to the corresponding method table. When this is done, the different classes are ready to be type checked.
For classes, the current class field is first set to the class that is currently being type checked. Afterwards the super class, the fields, and all methods of the class are type checked.
The type check method of `MJType` only checks class types and the base type of array-types. The type is checked by performing a lookup on the type name in the IR support class table. If no class with the same name is found, a type checker exception is thrown, informing about the lacking class.
The type check method of `MJVariable` is used to check the fields of the class. This method type checks the type of the variable in the same way that the type of the superclass is checked. In addition, if the variable is initialized when being declared, this initialization expression is also type checked. If the type of the expression is not assignable to the type of the variable, an expression is thrown. Types are defined to be assignable to each other if one of the following conditions are met:

- Both types are class types with the same name.
- Both types are class types and the source is a subclass of the destination.
- Both types are array types and their base types are assignable.
- The type of the types are equal and is neither a class type nor an array type (hence they are primitive types).
- The source is null and the destination is a class type.
- The source is a void type and the destination is a constructor.

After the super class and fields of a class are checked, all methods of the class are type checked. The type check method of `MJMethod` first assigns itself to the static current method field in the `IR`-class. The return type and all the parameter variables of the method are then type checked. A new scope for the parameters is also created and added to the scope stack of the `IR`-class. If the method is a constructor, a call to the default constructor of the super class

should be added if the body of the constructor does not contain a call to any constructor of the super class as its first statement. This constructor is found by using the class table of the IR-class and a constructor call is then added as the first statement of the body of the method. Afterwards the body of type MJBlock is type checked and the scope is left.
For type checking a MJBlock, all variables declared in the block are first type checked. All these variables are then added to a new scope similar to the parameters of MJMethod. Afterwards, all statements of the block are type checked and the scope is left.
The abstract MJStatement class inherits the typeCheck-method from the IR-class. It is therefore possible to use polymorphism for type checking all statements. Below is an overview of how the different subclasses of MJStatement are type checked.

- MJComment: There is nothing to type check in this class. The IR method is therefore overwritten with an empty method.

- MJAssign: The MJIdentifier on the left hand side and the MJExpression on the right hand side are both type checked. After this, it is checked whether the expression is assignable to the identifier.

- MJIfElse: The condition of type MJExpression is type checked, and it is ensured that the condition has the boolean type. Afterwards the then-block is checked and if an else-block is supplied, this block is checked as well.

- MJWhile: Similar to the MJIfElse-statement. The condition is type checked should be boolean, and the body is type checked as well.

- MJMethodCallStmt: All parameters of the method are first checked. If the method identifier is an instance of MJSelector, the object identifier is first type checked. This identifier must be a class type and an exception is thrown if this is not the case. The definition class is then set to the declaration class of the type and the method name is retrieved from the MJSelector. If the method identifier is not a MJSelector, it is checked whether the identifier name is equal to the "this" or "super" keywords. If it is, it implies a call to a constructor in the current class or the super class. If the name equals "this", the definition class and the name of the method is set to be the current class, and if the name equals "super", the definition class and method name is set to be the superclass of the current class. If neither, no definition class is set, and the method name is set to the name of the identifier.
  The class table in the IR-class is then used to retrieve the method. If the method or class is not found, a type checker exception is thrown informing

about this. Otherwise, the method is bound to the method call for use in the back end.

- `MJReturn`: The expression to be returned is type checked and the type of this expression is retrieved. The type is then compared with the return type of the current method by ensuring that the expression is assignable to the return type. Otherwise an exception is thrown.
- `MJPrint`: In this statement the parameter is type checked.
- `MJPrintln`: Same as `MJPrint`.
- `MJBlock`: The type checking of this class is already explained earlier in the description of the phase.

The abstract `MJExpression` also inherits the `typeCheck`-method. The overview below explains how the method used to override this inherited method is implemented:

- `MJBinaryOp`: All binary operators first type check the left and the right hand side expressions. When this is done, the different operators require that the left and right hand side have the same specific type.
  - Arithmetic operators only accept expressions of type integer and is an integer type.
  - Logical operators are used on expressions of type boolean and are boolean themselves.
  - Comparison operators normally accept any pair of expressions of the same type that can be compared. In this compiler, the `MJEqual` operator accepts boolean and integers, and the `MJLess`-operator only accepts integers. The expression is of type boolean.
  - String operators accept strings and are of type String.

  Note that the `MJPlus` operator is both used as an arithmetic operator and a string operator, thus this operator has to have arguments of the same type that are of either the integer or the String type.
- `MJUnaryOp`: Subclasses of this abstract class first checks the expression supplied. If the operator is a `MJNegate`, the compiler checks whether the expression is a boolean type. If the operator is a `MJUnaryMinus`-operator, the expression has to be equal to the integer type.
- `MJParentheses`: Type checks the expression inside the parentheses and returns the type of this expression.

- `MJIdentifier`: The type check method starts out by retrieving the corresponding `MJVariable` from the scope stack in the `IR`-class. If the variable is not in the scope, an exception is thrown. Afterwards the variable is saved, binding the identifier to the variable for later use. If the super keyword is used it can be inferred that the identifier is a `MJSelector` where the declaration variable does not matter, as the declared variable will be a field or a method.
- `MJIfThenElseExpr`: Similar to the `MJIfElse`-statement where the condition must be boolean. However, both the then-expression and the else-expression are required to have the same type.
- `MJNew`: The type of the `MJNew`-expression along with all argument expressions supplied are type checked. Afterwards the constructor is found by using the class table in the `IR`-class and binding this to the `MJNew` statement.
- `MJMethodCallExpr`: First it is checked whether the method identifier is an instance of `MJSelector`. If it is, the object part of the selector is type checked and the corresponding class is found and used as the declaration class. The field part of the `MJSelector` is used as the method name. All argument expressions supplied to the method are then type checked and the method is found by using the class table and bound to the method call. If no method is found an exception is thrown.
- `MJCast`: The type of the cast and the identifier to cast are both type checked. Then it is checked whether the identifier can be cast to the cast type by checking whether the type of the identifier is assignable to the cast type.
- `MJNoExpression`: No type checking is necessary here.
- `MJBoolean`: Sets the type of the expression to the boolean type.
- `MJInteger`: Sets the type of the expression to the integer type.
- `MJNull`: Sets the type of the expression to the null type.
- `MJString`: Sets the type to the String type and type checks this type.

This concludes how all parts of the IR are type checked.

The next phase is the *variable initialization check phase*, where it is checked whether the program requests variables which have not been initialized when requested. The traversing order is the same as in the type checking phase except that the `MJType`-class is not checked for variable initialization in any class.

The program calls variable initialization on all classes. Each class then adds all its fields to a set that contains all initialized variables. The "this"-field is also added to the set. The class then does a variable check on its methods, passing the set with initialized variables as argument.
In the `MJMethod`, the parameters of the method are added to the set and the `MJBlock` of the method is checked. Here, all variables declared in the beginning of the block are checked followed by checking all statements of the block. A `MJVariable` is checked by examining whether the variable is initialized when being declared. If it is, the variable is added to the set of initialized variables.
As in the type check phase, the different subclasses of `MJStatement` implement the variable initialization check in different ways. The overview is seen below:

- `MJComment`: There is nothing to check in this class. The IR method is therefore overwritten with an empty method.
- `MJAssign`: The right hand side expression is checked first followed by the identifier on the left hand side. It is important that the right hand side is checked before the left hand side, as the identifier on the left side might be used on the right hand side. If the left hand side is checked first, the variable will be added to the set of initialized variables, making the right hand side variable check pass even though the variable might not have been initialized.
- `MJIfElse`: The condition is checked for variable initialization. Afterwards, the then-block is checked with a copy of the set - variables only initialized in the then-block should not be added to the initialized set outside the then-block, as this block might be skipped. The same check is done on the else-block if it is present, and the intersection of the then-set and the else-set is computed and added to the initialized variables set. This is done because the variable will be initialized after the `MJIfElse`-statement if it is both initialized in the then-block and in the else-block.
- `MJWhile`: Similar to the `MJIfElse` without an else-block.
- `MJMethodCallStmt`: All arguments are checked and if the method identifier is a `MJSelector`, it is checked that the object used for calling the method is initialized.
- `MJReturn`: The return expression is checked for initialization.
- `MJPrint`: The parameter expression to the print statement is checked for initialization.
- `MJPrintln`: Same as `MJPrint`
- `MJBlock`: This is explained above.

The overview of the subclasses of `MJExpression` is seen below:

- `MJBinaryOp`: Checks the right hand side and the left hand side for variable initialization.
- `MJUnaryOp`: Checks the argument for variable initialization.
- `MJParentheses`: Checks the expression.
- `MJIdentifier`: If the identifier is the left hand side of an assign statement, the variable bound in the type check phase is bound to the set of initialized variables. If not, it is checked whether the variable bound to the identifier is contained in the set of initialized variables. If this is not the case, an exception is thrown, as this means that an uninitialized variable is being used.
  If the identifier is an instance of `MJArray`, the index is checked as well. Note that individual elements of the array are not checked.
- `MJIfThenElseExpr`: Checks the condition and the then-expression and else-expression.
- `MJNew`: Checks the list of arguments.
- `MJMethodCallExpr`: The same as `MJMethodCallStmt`.
- `MJCast`: Checks the identifier to be cast.
- `MJNoExpression`: No checking is necessary here and the method is therefore empty.
- `MJBoolean`: Same as `MJNoExpression`.
- `MJInteger`: Same as `MJNoExpression`.
- `MJNull`: Same as `MJNoExpression`.
- `MJString`: Same as `MJNoExpression`.

Notice that the only place where it is possible for the compiler to fail is in the `MJIdentifier`-class, as this is the only place where uninitialized variables might be read.

The last part of the analyse phase is the *second rewrite phase*. This phase traverses the IR tree and converts all `MJIdentifier` objects which are fields into `MJSelectors`, where the current class is set as the defining object and the original identifier is set as the field identifier of the new selector.

The IR is now finished being analysed and is sent to the code generator.

2.2.2.4 Back end

The back end is responsible for converting the IR into the target language. The target language for the original compiler is the LC3 assembler language. The specification for this processor is found in the ZIP-archive handed in with the project, see Appendix D.
Initially a `CODE`-class is created. This class contains an initially empty list of instructions, fields corresponding to the eight registers of the LC3 processor, and labels pointing to different commonly used operations that should not be written more than once. The `CODE` class also contains some useful operations such as pushing to and popping from the stack memory, commenting, creating labels, and of course printing all lines of code to an output file.
The stack memory is maintained by using one of the registers as a stack pointer, denoting the top of the stack. When data is pushed to the stack, a STR-instruction of the LC3 is used which writes to a memory location specified by the stack pointer, and when data is popped, the LDR-instruction which reads from a memory address specified by the stack pointer is used. When a method is invoked, the compiler makes space for a *stack frame* and the *stack frame pointer* maintained in one of the registers is set to point at the old stack pointer, which will be the beginning of the stack frame. Within the stack frame, there is made space for the arguments of the method and for local variables. All arguments of the method are also pushed into the stack frame. These, and any local variables declared, can now be retrieved by using the stack frame pointer and an offset.
The heap is more complex. When an object is created using the `MJAssign`-statement, the right hand side first generates instructions. This `MJNew`-expression calculates the size of the object that is to be created by calling `getSize` on the corresponding class. The size is then pushed onto the stack and the Program Counter (PC) then jumps to the new-routine added to the list of instructions in `MJProgram`. This routine retrieves the required memory size from the stack and creates a new heap pointer where the object can be stored and pushes this pointer on the stack. The left hand side of the assign statement pushes the stack frame address of the variable to the stack and the `MJAssign` then pops the stack twice and saves the heap address in the variable address. When an object is read from the heap, the heap pointer is retrieved from the memory address specified by the identifier.
In the second rewrite phase of the analysis part of the compiler, all identifiers referring to fields were rewritten to be `MJSelectors` instead. An object of this type will therefore always be used when reading and writing fields of a class. First, the heap address of the containing object is retrieved. If the object address is zero, a null pointer exception is thrown, as this implies that the object is null. If not, the offset of the variable is retrieved and this offset is added to the heap address, resulting in the address of the field. This address is pushed onto the stack if the field is being written to. The `MJAssign`-statement will then

write the right hand side to the address of the field. If the field is being read, the value stored at the address of the field is then pushed onto the stack, ready for being used in an expression.
Methods are handled by creating a new stack frame when being called as explained above. The result of the method (if any) is placed on top of the stack. Each method is also assigned a `LC3Label`, which is used to find the requested method. The instructions of the body of the method are then generated after this label and given a return statement.
All instructions used by the compiler have a corresponding class in the `compiler.CODE.LC3`-package. An abstract `LC3`-class is also defined which all instructions inherit from. This class defines an abstract `toString`-method, which each instruction implements. The method is used to formulate how the class is turned into a textual machine instruction. The list of machine instructions in the `CODE`-class contains objects of type `LC3`, making it easy to use polymorphism to iterate through the list and call the `toString`-method on each instruction in order to create the full assembler instruction set.
The code generation of all `IR`-classes will not be explained in this report. Most of the `MJExpression`-subclasses have a corresponding machine instruction. However, the control flow statements of MiniJava, the `MJIfElse` and `MJWhile` statements, are explained below.
The `MJIfElse` first generates code for the condition expression. The result of this expression is popped from the stack and if the value is zero (hence false), a jump is made to the label defined after the then-block. If an else-block is defined, the processor executes the instructions of this block, which will be generated right after the false-label. If the condition evaluates to non-zero (hence true), the PC continues and execute the then-block. If an else-block is present, the PC jumps to the label defined after this block, thereby skipping it.
`MJWhile` also generates code for the condition first. The value is popped from the stack and if non-zero, the PC continues and execute the code generated for the body of the while statement. When the body is executed, the PC jumps back to the condition and evaluates it again, creating the loop. If the condition evaluates to zero, the PC jumps to the end of the while statement and continues execution.

2.2.3 Making the compiler modular

Now that the three phases of the compiler have been analysed, it is possible to begin to look at how to separate the different modules into different programs. In order to do this, it should first be checked that the different modules of the compiler are loosely coupled and easy to separate.

2.2.3.1 Internal modularity

External modularity is easier to achieve if it is first made sure that the compiler is internally modular. In that way, it is possible to move the module into a new program without making significant changes.
As mentioned earlier, the compiler used in this project is split into three phases. However, the compiler does not follow the architecture implied in figure 2.1, where the analysis and code generation are done in a separate module. Instead, the different IR-operations are defined in the IR itself. This makes it more difficult to implement new back ends, as every element in the IR would need a new operation for code-generation. Furthermore this implementation also makes the IR more messy, as analysis operations and back end operations are in the same class. Finally, when separating the different modules, a different IR-datastructure has to be provided for each module, as operations related to one module should not be present in another module e.g. back end operations in the front end module.
A solution to these problems would be to use the visitor design pattern[2]. The pattern basically extract methods from different classes and group them together in one class. It would then be possible to place all analysis operations in one visitor and all code generation operations in another, effectively encapsulating both phases. Even pretty printing could be put into a visitor, making the IR even more tidy. Additionally, if a new back end is needed, one could provide a visitor with the desired code generation. This would also make it easier to separate the different parts of the compiler into different modules.
However, this approach was used for the compiler in earlier versions and was removed again because the users had trouble understanding the visitor pattern, and too much time was used on understanding and learning how to use this pattern. Therefore, the visitor pattern-approach was not well-suited for teaching purposes, although one could argue that the visitor pattern is part of learning about compilers, as this is a very frequent use of this pattern. When separating the modules, it is therefore necessary to provide different operations for the IR-classes for the different phases.

2.2.3.2 External modularity

In order to separate the modules, not only internally but also into different programs, each module has to read an input file, process it, and write a file which is either the result or the input for the next program. The file passed from one part of the compiler to the next has to contain all necessary information in order for the next part to continue the compiling process. All this information is contained in the IR, so a file format for effectively storing the IR is needed. As

the IR has a tree structure, it seems appropriate to select a serialization format which is also structured like a tree. Furthermore, the IR should be human readable, both for debugging purposes and for the user to be able to investigate the IR between each module.
Two serialization formats that fulfill these requirements are XML and JSON. For huge tree structures, XML is generally more readable because of the closing tags. In XML, it is also easy to add meta data as attributes, separating this from the actual data. For these reasons, XML is chosen as the file format for the IR and the flow for the whole compiler is seen in figure 2.4, where an XML-file is being transferred between each module.

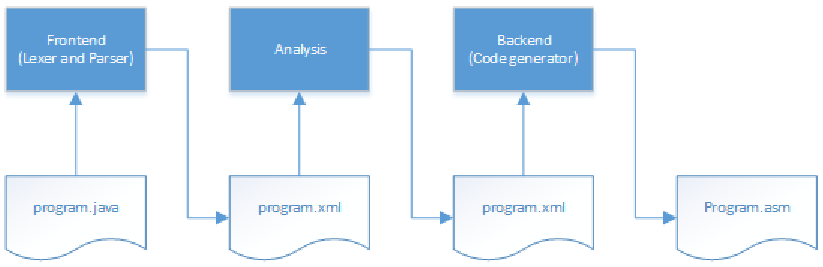


Figure 2.4: The overall structure of the compiler

Several approaches to writing the IR as XML exists:

- Using an XML-library to write and read the XML for the IR.
- Writing an operation for each element in the IR that specifies how that element should be serialized. This method would also have the responsibility of serializing the subelements of the element.
- Writing an operation for each element in a separate visitor.
- Writing a serializer which is completely separated from the IR using reflection.

At a first glance, using a library seems most appropriate, as this solution would not require time to create and debug a serializer. The most obvious choice would be to use JAXB, a library included in the standard libraries of Java. However, this library along with all other XML-libraries is dependent on having a list of classes that the serializer should know about and either require annotations on fields, or public setters and getters for all fields. As changes to the original compiler should be avoided and the IR should be kept clean of anything related

to serialization for teaching purposes, this might not be the best option. For the same reason, writing all serialization logic in the IR is also a very inflexible solution. If anything is to change about the IR, the serialization would also have to change. Writing a visitor is also very inflexible, as this visitor would have to know every IR-class and also need to have access to all fields of the IR.
The last option is to use reflection to create the serializer. Not only would this make it possible to access and write to private fields, the serializer would not need to know anything about the IR at all if designed properly. Furthermore, it would not be necessary to change much in the existing program, hence the users would be well-separated from the serialization logic.
Reflection is by some considered bad practice in programming as encapsulation is broken and the performance in general is slow[3]. In this case performance is not an issue, and the alternative would be to create public getters and setters. By using reflection, the fields are still private when the users are using them. In addition, as all fields will be treated as a key (the field name) with a corresponding value (the field value) of a specific type (the field type), the serializer does not know anything about which class it is serializing. Therefore, the encapsulation is not broken, as all fields are just treated as data to be serialized. The Java library mentioned above also uses reflection[4].
One of the consequences of making the compiler externally modular is that all fields in one module must have the same names as the corresponding fields in the next module. If not, the serializer would not know what field it should deliver the data to. This is of course a problem when the compiler is going to be used for teaching purposes, as all fields would need to be present in advance, or the students would need to know what they should call their fields. An alternative would be to use the order that the fields are in, so that the first field in one module corresponds to the first field in the next module. However, then the users implementing the compiler would need to know what order to define their fields in, which is hardly better and probably also more confusing. After discussion this with Associate Professor Christian Probst, it was agreed that a theoretical exercise prior to the implementation would make it acceptable to define the fields for the users. In this exercise, the users will define the fields that are necessary in the different IR-elements, and when the program skeleton is handed out, all fields will be there in advance.

2.3 Design

In this section, a model of the persistence layer is designed and described. By creating a model, the output of the serializer is able to be read by the deserializer as long as both modules are using the same model.

2.3.1 Defining a formal syntax for the XML

Before the serializer can be created, a formal syntax for the XML should first be defined. This will define the legal building blocks of the XML-document and ensure that all modules work with the same XML-elements and attributes.
In the IR datastructure, each field is one of following:

- A primitive such as a string or integer.
- An enumerable collection such as a list.
- A map.
- Another IR object.

The formal syntax should therefore support the above type of fields.

Another problem that should be addressed is the fact that references are lost when objects are serialized. When deserialized, all fields will contain different instances of classes. This will create errors various places in the program. The class table in the `IR`-class is one such example. When changes are made to a class in the classtable, the corresponding class in the IR-datastructure will not change, which will lead to errors. Therefore, the references need to be serialized too. This can be done in XML by adding an ID-attribute to each object. The ID can be used to define which instance the object belongs to by letting all other objects, belonging to the same reference, have the same ID. When deserializing, the deserializer has to make sure that the same instance is used when assigning fields with the same ID.
As every instance only needs to be serialized once, there is no reason to serialize an object unless its ID has not occured before. When an ID occurs in an XML-element a second time, the content of the element is redundant and should therefore be empty. However, for improved readability, a placeholder is introduced, informing the reader that the content is already serialized earlier in the document.

As the element name on many of the XML-elements will be variable, neither the Document Type Definition (DTD)[5] nor XML Schema (XSD)[6] notations can be used for this formal syntax. Instead, a more general context free grammar is used, namely BNF, which is also used to define the grammar of MiniJava. The BNF is seen in figure B.1 in Appendix B.
As the syntax implies, both objects, collections, and maps have ID-attributes. Furthermore, the `<ALREADYSERIALIZED>`-tag defines the placeholder for objects which are already serialized, and is only contained in the grammar for improved readability. The `<CLASSNAME>` must denote an existing class, either in the current program or in the java library, and `<FIELDNAME>` must denote a valid field for the `<CLASSNAME>` that the tag belongs to. The `<COLLECTIONCLASS>` and `<MAPCLASS>` tags must denote the concrete classes that were used as a collection or map.

2.4 Implementation

In this section the implementation of the serializer and deserializer is discussed and explained along with the separation of the compiler into different Java projects. Furthermore, various implementation issues as well as the chosen solutions to these are examined.

2.4.1 Structure of the serializer and deserializer

The implementation consists of two parts: Creating the serializer and creating the deserializer. The job of the serializer is to write the formal syntax in figure B.1 found in Appendix B while the deserializer should read it. These two processes should be separated into different classes for two reasons. First of all, both the serializer and deserializer will contain a lot of code, making it more tidy to put each in its own class. Secondly, not all three modules need both a serializer and a deserializer:

- The front end module only needs a serialize.
- The analysis-module needs both a serializer and a deserializer.
- The back end module only needs a deserializer.

When the two parts are in different classes, the above requirements are easy to satisfy. However, this will also introduce some redundancy, as the serializer and deserializer are sharing some constants (like the `<ALREADYSERIALIZED-`placeholder) and utility methods. To minimize this, an utility class is introduced, containing these constants and methods. This class, along with its relation to the serializer and deserializer, is seen in figure 2.5, where two constants and two methods are contained in the class.

2.4.2 Serializing

The serializer should be able to transform the IR-datastructure into the XML syntax defined in figure B.1 found in Appendix B. According to this grammar, the XML needs to start and end with a tag containing the name of the class that the object being serialized is an instance of. The name of this class is retrieved using reflection, along with the id from the id-counter which will be zero at this point. The starting and ending tags are then added.

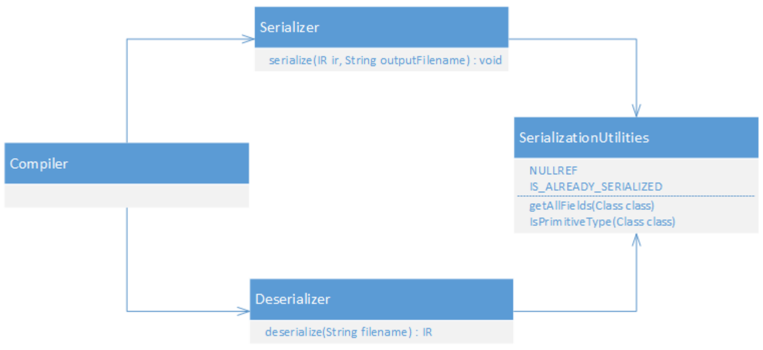

Serializer
serialize(IR ir, String outputFilename) : void
SerializationUtilities
NULLREF
IS_ALREADY_SERIALIZED
getAllFields(Class class)
IsPrimitiveType(Class class)
Compiler
Deserializer
deserialize(String filename) : IR

Figure 2.5: A class diagram illustrating how the serializer and deserializer are related.

The method `getXmlString`, which takes an object as an argument, now serializes the IR-object. This is accomplished by serializing all the fields of this argument. To do this, the serializer first needs to have a way of accessing these. Many of the fields that are to be serialized are encapsulated. Furthermore, not all of these fields have getters and setters. Even if they did, writing the serializer would be very cumbersome, as it would not be possible to treat every IR-object the same way. Different getters would have to be called for each objects. A way around this, which also allows the serializer to treat every IR-object the same, is reflection.
One of the methods of the `Object`-object, is the method +`getClass`, which returns the class of the object. Afterwards it is possible to extract all declared fields, including private fields from this class. This is accomplished by using the `getAllFields`-method defined in `SerializationUtilities`. Next, the serializer loops through the fields and set them to accessible by using the `Field` method `setAccessible(true)`. It is now possible to read and write these fields by supplying the object that the fields are going to be read from. Note that no changes are made to the actual fields of the class, but only to the `Field`-instance.[2] The classes are therefore still encapsulated.
Now that the serializer is able to loop through the fields, these can be serialized. An opening tag is first added, where the tag name will be the name of the field. Next, a tag containing the name of the class that the field holds is created. The name of the class is read from the object contained in the field, if possible. If the object is null the method`getClass` cannot be called from it. However, the formal syntax defines that a tag with the name of the contained class instance is needed. The name of the class which is *allowed* to be contained in the field is therefore extracted by calling `getType` on the `Field`-object instead as this

[2]"A value of true indicates that the reflected object should suppress Java language access checking when it is used" - taken from the official Java documentation[7].

will never be null. The name of the resulting class is then used. The reason for not using the latter approach in both cases is that this might not return the actual class of the object, but rather a superclass. If the type of the field is `Object`, it could hold any type of object deriving from this class. However, this information is not important if the object is null, as no class is needed to instantiate from.
The class name tag also needs an ID. In order to keep track on which objects have already been serialized, a map is created in which every contained object has a corresponding ID. Initially, the `HashMap` implementation was used as only dictionary operations are going to be used (add, delete, contains). This resulted in issues later during testing, as this implementation only checks whether the objects return the same hash code and are equal during a lookup. The consequence of this is that two references end up referring to the same object when being deserialized, although the two references originally referred to two different objects. The `IdentityHashMap`-implementation was therefore used instead. This class is implemented by checking if the objects have the same reference using the ==-operator. Furthermore, the `IdentityHashCode` of the objects are used instead of any overwritten hashcode-function[17]. If the object which is currently being serializing is contained in the map, the ID of the class name tag is set to be the ID already given the object, and the `alreadySerialized`-flag is set to true. If not, the ID-counter is incremented and a new ID is assigned to the object and added to the classname-tag. This information is afterwards saved in the map. If the object is null no ID is needed. A field is seen serialized below:

```
<compiler.IR.IR id=0>
    <program>
        <compiler.IR.MJProgram id=1>
         ...
        <compiler.IR.MJProgram>
    </program>
</compiler.IR.IR>
```

Next, if the `alreadySerialized`-flag was set to true, the `IS_ALREADY_SERIALIZED`-placeholder string is inserted as the content of the tag. Otherwise the `getXmlStringGeneral`-method is called, which returns the XML-content for the object. This method depends on the type of the object:

- If the object is null the string constant `NULLREF` is returned. This is used instead of the string "null" as this string might be used elsewhere, especially in a compiler.

- If the object is of a primitive type (one of the wrapper classes of Java or the class `String`), the result of the `toString`-method is returned.
- If the object derives from `Map` the object is cast to a map, and the overloaded version of `getXmlString`-is called recursively with the object as parameter and the result as return value.
- If the object derives from `Collection`, the object is cast to a collection and the overloaded version of `getXmlString`-is called recursively with the object as parameter and the result as return value.
- If none of the above recursively call `getXmlString` with the parameter type `Object` and return the result.

In the early stages of this serializer all IR objects, including the support classes, had a common interface, `IRBase`. This interface was empty, and the only purpose of the class was to limit the objects accepted for serialization. The serializer was later refactored, as it was noticed that the method used for serializing IR objects could be used to serialize any object. With the addition of circular references support, the serializer was more than fit to serialize arbitrary objects. The overloaded method `getXmlString` taking a parameter of type `Collection` iterates over the collection, giving each object in the collection an ID and serializing it by calling `getXmlStringGeneral` on it. The result is seen below:

```
...
<java.util.LinkedList id=23>
    <compiler.IR.MJMethod id=24>
        ...
    </compiler.IR.MJMethod>
    <compiler.IR.MJMethod id=56>
        ...
    </compiler.IR.MJMethod>
</java.util.LinkedList>
...
```

Similarly, the corresponding method taking a parameter of type Map iterates over all keys and serialize these along with their corresponding mapped value, giving both ID's if needed. The key and value are grouped together with a `KeyValuePair`-tag as seen below:

```
...
<java.util.HashMap id=125>
```

```
    <KeyValuePair>
        <Key>
            ...
        </Key>
        <Value>
            ...
        </Value>
    </KeyValuePair>
</java.util.HashMap>
...
```

The serializer is now able to turn the entire IR into an XML-string. However, at this point there are still some special cases that should be taken care of.
First of all, circular references have to be considered. A circular reference exists if a cycle is present in the object hierarchy. The recursion might never end if there is a circular reference, as the program will be stuck in a recursive loop until the program runs out of memory.
The IR-datastructure is a tree, and should in theory not have any circular references. However, after debugging the IR, two circular references are found. The first one exist because the superclass of the MiniJava-class `Object` is set to be `Object`. The second circular reference is because the MiniJava-class `String` have a field `text` with the type String.
The first problem is easy to deal with. The superclass of Object is changed to the null-value, and a boolean field is added, indicating whether or not this class is the root of the class hierarchy. The second problem is more difficult. In a normal compiler, the text-field would be of type `char[]`, but this approach does not work here as the compiler does not support characters as a primitive value. Initially, the field is just commented out of the program as it was not assigned by the compiler anyway and therefore rather useless. Later when circular references are implemented (see below), the field is uncommented again.
Another type of circular reference is those caused by some static fields. In some objects in the compiler, a static field is assigned an object with the same type as the class which the field belongs to.
In order to handle this, and also decrease the chance for future errors, the serializer and deserializer should be adjusted to support circular references. At this point the serializer can already handle circular references due to the ID-attribute. When the serializer hits an object which is already serialized, or is being serialized, it stops and uses the `IS_ALREADY_SERIALIZED`-placeholder as content for the tag. The result of serializing a circular reference is seen below:

```
<compiler.IR.MJBoolean id=42>
    <value>
        <compiler.IR.MJBoolean$MJBooleanValues id=43>
```

```
            True
        </compiler.IR.MJBoolean$MJBooleanValues>
    </value>
    <False>
        ...
    </False>
    <True>
     <compiler.IR.MJBoolean id=42>
            IS ALREADY SERIALIZED
        </compiler.IR.MJBoolean>
    </True>
</compiler.IR.MJBoolean>
```

This is why only the deserializer has to be changed in order to support circular references, which will be covered in the next section.
Another special case is enums. No changes have to be made, but it should be clarified that the classname for an enum consists of the class that it is contained in followed by a $ and the enum name. An example is
`compiler.IR.MJType$TypeEnum`.
Finally, the `getDeclaredFields` called on a class to get the fields does not include fields defined in its superclasses. This is a problem for classes deriving from `MJBinaryOp` for example, as the right-hand side and left-hand side expressions are both declared in the superclass. To solve this, a recursive method was created in `SerializationUtilities`, which also gets fields from the superclasses. This recursion is stopped when either the `IR`-class or the `Object`-class is hit, as the fields for the `IR`-class is not needed in subclasses. In order to support overwriting of fields, a field defined in a superclass is ignored if the class contain a field with the same name.

2.4.3 Deserializing

Now that the whole IR is able to be written into an XML-file, the next step is to create a deserializer which is able to read the file and recreate the IR from it.
First, a line counter variable is initialized, which is used to tell which line of the file is currently being deserialized, and a method is created which reads a line from the file and increases the counter. This is useful for exception messages and debugging purposes, and the file will be read only through this method. A `Scanner`-object is used for parsing the xml-file.
The first line is passed to the `readXml`-method. This method processes the line by validating and reading the tag which contains a classname. This classname is used to create a `Class`-object, which can be used to instantiate instances. If

the tag contains an ID-attribute, this is read too. The content of the tag is then deserialized. As in the serializer, this can be either a map, a collection, a primitive, or an object. It can also be an enum, which has to be treated differently from an object when deserializing.
When reading an object the deserializer first checks whether the content of the tag equals the `NULLREF`-placeholder in `SerializationUtilities`. If it does, the XML-tags are validated and null is returned. Otherwise it is known that the content is an object of the type specified by the enclosing tags. Using reflection, a new instance of the class read from the XML is created, and a list of `Fields` from this class is retrieved using the helper operation defined in `SerializationUtilities`. All these fields are also set to accessible in order to make it possible to assign them. Next, the names of the fields listed as content of the class-tag are read, and a matching field is found in the list of fields of the instantiated class. If none is found, an exception of the type `XmlParseException` is thrown as this is an error in the XML. Afterwards, the content of the field is read by recursively calling the method `readXml`. The object returned by this method is then assigned to the field using reflection. Fields are being read like this until a closing tag appears for the object that the fields belong to.
It is important to note that not all fields of a class need to be in the XML. If a field is missing in the XML, the corresponding field in the class will be left unmodified by the deserializer. A consequence of this is that it is possible to safely add new fields to the IR-classes without these being modified. On the other hand, removing fields from the class that are in the XML is not possible with this implementation. It would be possible to modify the deserializer to skip the field if it was not in the class, but as the program is used for teaching purposes and all the fields that the users should use are present, it should not be necessary to add more fields which are not present in later modules. If the users do this, the exercise have probably been solved in a wrong way that would not work with the other parts anyway. Therefore, this check ensures that the user does not add unnecessary fields.
Another thing to be noted is that it is not possible to instantiate a class through reflection if it does not have a parameterless constructor. Not all the classes in the IR have this, thus one would either have to be added, or each class would have to be treated separately. As the first option is the most flexible and the easiest one to implement, this was chosen. This also means that a parameterless constructor is added to all existing IR-classes and that any new classes belonging to the IR should have one too. If the class read from the XML is a primitive type, an instance of that type is assigned to the field. In order to do this, the class that the primitive belongs to is identified, and an instance of the corresponding wrapper class is instantiated. If the class belongs to the String-class, the content might be equal to the `NULLREF`-placeholder. If this is the case, null is returned.
A problem is encountered when the type of the class derives from `Map` or `Collection`. Both of these classes have generic type arguments. In Java, gener-

ics are implemented by type erasure[15], which erases all information about generic types at compile-time. A `Collection<String>` will therefore end up being an instance of the raw type `Collection`. As a consequence of this, generic type information is not usable at run-time. The main reason for this implementation according to the official Java documentation[15] is to make generic code compatible with legacy code. Other languages like C# implement generics that are readable at runtime[16], which is a very powerful tool when working with reflection.
The consequence of this implementation is that reflection can not be used to read the generic type arguments from generic collections and maps. Even if it was, these would not be of much use, as another consequence of type erasure is that it is not possible to instantiate any generic class at runtime. Instead, all generic type parameters are set to the `Object`-type, making it possible to create a non-generic map at runtime. The disadvantages of this is that it is possible to put any kind of object into the map/collection before it is added as value to a field, which will result in a run-time error at some point when the program attempts to cast this object into the generic type. This is only a potential problem in the deserializer, however, as any attempt to put another object in the map/collection defined in the IR will still result in a compiler error.
When an instance of a collection has been created, the deserializer keeps reading objects until it hits the closing tag of the collection, after which the constructed collection is returned. Maps use the same approach for reading except instead `KeyValuePairs` are read until the closing tag of the map is hit. Each `KeyValuePair` consists of a key object and a value object which are put into the instantiated map.
Finally, if the class is an enum, all possible enum values from the class are retrieved, and the deserializer iterates over these. If the value between the tags of the class matches any of these, the enum value that matched is returned. A parsing exception is thrown if no matching value was found.
A flowchart illustrating the relation between the different methods in the deserializer is seen in figure 2.6. A collection calls the `readXml`-method recursively on all element, a map calls the `readXml`-method recursively on all keys and values, and an object not matching any of the conditions call `readXml` on all contained fields.

In order to restore the references implied by the IDs of the XML, two maps are monitoring the deserialization process. One contains objects that are already deserialized, and the other contains objects that are currently being deserialized. An object is added to the `objectsBeingDeserialized`-map before being deserialized. When the object is fully deserialized it is removed from this map and added to the `deserializedObjectsMap`. Both maps use the ID of the objects as key. When a starting tag of a class is read, the deserializer checks if the object is currently being deserialized. If it is, the deserializer has hit a circular reference, and the `Field` which is currently being serialized and the instance that the field is belonging to are therefore saved. These data are saved in a container,

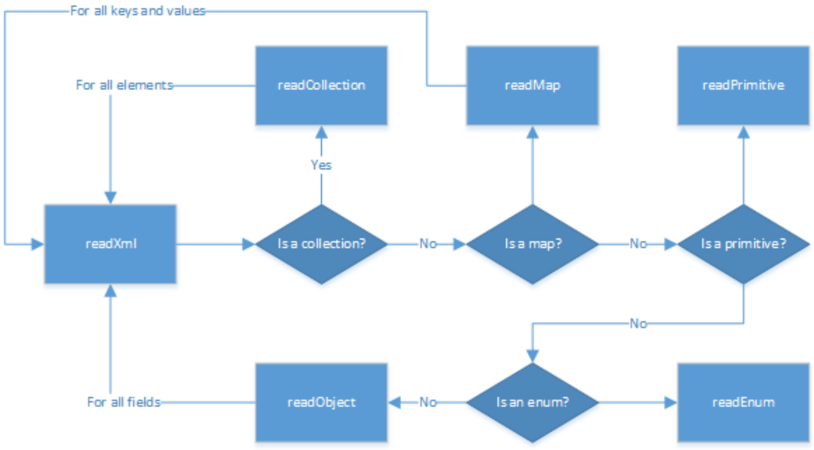


Figure 2.6: The main flow in the deserializer. The method readXml is the first to be called.

`ObjectFieldContainer`, as this needs to be kept together. The container is saved in a list as one object might have more than one circular reference. By doing this, the reference can be recreated by assigning the deserialized object to the saved fields and instances once the IR has been deserialized.
If the circular reference occurs in a collection, a different approach has to be taken. First, the order of insertion in a list should not be changed, so a global variable `collectionIndex` is used for remembering the position that the object currently being deserialized should be inserted at. If a circular reference is hit, the current value of `collectionIndex` is saved along with the list that the value should be inserted in. The `ObjectFieldContainer` is used for this too, but in this case the `field`-field is not used. Instead, an integer-field is added and used for the collection index. An additional integer-field containing the type of the container is also introduced, along with three integer constants, `FIELD`, `COLLECTION`, and `MAP`, which are the possible types. By using the type-field, it can be determined which fields to use from the container.
When an object is deserialized, a list is obtained from the `objectsBeingDeseria-lizedMap` by using the id of the deserialized object as key. This list of `ObjectFieldContain-` \verb
represents all circular references contained in this object. If the container is of type `FIELD`, the `Field` stored in the `field`-field and the instance stored in the `object`-field can be used to assign the deserialized object to the field. If the container is of type `COLLECTION`, the deserializer first has to check whether the collection implements the `List`-interface. This is because the order matters for a list, whereas the order for a set is undefined. As the class hierarchy for the `Collection`-collection contains no other data structures with a defined order[18], the deserializer only needs to check for this interface. If the collection

is a `List`, it is possible to cast it and use the `add(int index, Object element)`-method declared by the `List`-interface to place the deserialized object at the correct index. If the collection is not a list, the `add`-method defined by the `Collection`-interface is used instead.
Circular references is not implemented for maps. The only maps being used in the compiler are in the compiler.IR.support package, which are the last to be listed in the XML. As these maps almost exclusively contain objects already stored in the IR, these have already been serialized, circular references should not be a problem. No problems with circular references in maps were found using the test script written in section 2.5 either. If this were to be a problem, however, it would be possible to implement by saving the key and value and store these in a container similar to what was done for collections.
If the object is not currently being deserialized, it is checked if it was already deserialized. If it was, the deserialized object is returned, which will recreate the reference as it was in the original IR.

2.4.4 Separating the phases into different projects

Now that a functional serializer and deserializer has been implemented, it is possible to separate the different modules of the compiler into different projects. Three new Java projects are therefore created, and are all described in this section.

2.4.4.1 Front end

The first project, which is to be the compiler front end, is supplied with the ANTLR-library required in order for the ANTLR-generated parser to work. The following classes from the packages listed below are copied into the new project from the compiler:

- `compiler`
  - All classes
- `compiler.Exceptions`
  - `CompilerError`
  - `ParseError`
- `compiler.Frontend`

  - All classes

- `compiler.IR`
  - All classes
- `compiler.Phases`
  - `Frontend`

Notice that the support classes of the IR have been removed. As the custom serializer allows new fields in later phases, all fields containing support classes have been removed, as these are not used before the analysis phase. A lot of the exception classes have also been removed, as these are not used either, and the phases that are not used have been removed from the `compiler.Phases`-package. In addition, the new `Serializer` and `SerializationUtilities`-classes have been added to the `compiler`-package.
All rewrite, type checking, variable initialization checking, and code generating methods have been removed from the IR classes. The LC3-label defined in each `MJMethod` has also been removed, as this field is only used in the backend. The main compiler class has also been edited so that only the front end is called, and a call to the serializer has also been inserted.

2.4.4.2 Analysis

The following classes from the packages listed below are copied into the new analysis project from the compiler:

- `compiler`
  - All classes
- `compiler.Exceptions`
  - All classes except for `ParseError` and `CodeGenException`
- `compiler.IR`
  - All classes
- `compiler.IR.Support`
  - All classes
- `compiler.Phases`

  - `Analysis`
  - `Rewrite`

The new `Serializer`, `Deserializer`, and `SerializationUtilities`-classes have also been added to the `compiler`-package.
All code generating methods have been removed from the IR classes. The LC3-label defined in each `MJMethod` has also been removed, as this field is only used in the backend. The main compiler class has been edited with a call to the deserializer to retrieve the IR. The call to the code generation phase has been removed and a call to the serializer has also been inserted. Furthermore, the `XmlParseException` has been added to the exception package.

2.4.4.3 Back end

The new back end project contains the following classes:

- `compiler`
  - All classes
- `compiler.CODE`
  - All classes
- `compiler.CODE.LC3`
  - All classes
- `compiler.Exceptions`
  - `ClassNotFound`
  - `CodeGenException`
  - `CompilerError`
- `compiler.IR`
  - All classes
- `compiler.IR.Support`
  - All classes
- `compiler.Phases`
  - `Analysis`

  - `Rewrite`

The new `Deserializer` and `SerializationUtilities`-classes have also been added to the `compiler`-package.
As the compiler is using the `MJClassTable`-support class in the back end to locate the `String`-class, the `ClassNotFound`-exception is still required in the `compiler.Exceptions`-package. All rewrite, type checking, and variable initialization checking methods have been removed from the IR classes. The main compiler class has been edited with a call to the deserializer to retrieve the IR. The calls to the other phases has also been removed. Furthermore, the `XmlParseException` has been added to the exception package.

2.5 Test

In this section the methods used for testing and debugging will be discussed and explained. A few errors discovered in the compiler will also be examined and fixed.

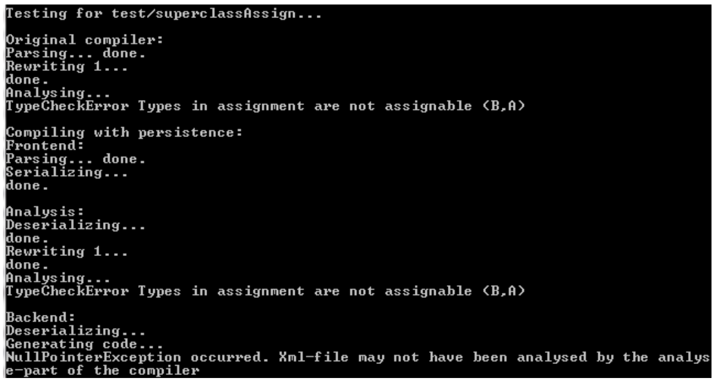
2.5.1 The approach used for testing

In the new compiler a layer of persistence has been introduced between each module. This, and the fact that the layer is being maintained using a custom serializer and deserializer, makes testing of the compiler vital. No behaviour should have changed after these changes were introduced, which should be tested for. Usually, this would be done by generating a number of test cases and check if these behave as expected. However, the assembler code that results from the compilation process is targeted a specific processor, the LC3, hence testing whether the code works is difficult without the LC3-processor, and an assembler belonging to the LC3 would be needed as well. Hence, a different and easier approach is taken.
When the compiler was received from Associate Professor Christian Probst, 21 test files were included with test programs. These programs cover all features of the compiler except for some special cases. By assuming that this compiler is working correct and generates error-free assembler code, all these test files can be compiled using the original compiler, generating correct assembler files. Afterwards, these files can be compared with the output of compiling the same test files using the modified compiler. If there are any differences between any pair of files, errors have been introduced.
In order to do this effectively, a tool for comparing files and which can be accessed from the command line is needed. By having access from the command line, test scripts can be written later. The tool FileEqualityChecker included in the attached archive (see Appendix D) was used for this. It is a simple tool written in Java which reads through two files and compare each line in the first file with the corresponding line in the second file. If there is a difference, an error along with a line number is returned. If not, a message confirms that the files are equal.
In order to effectively test the files, a batch file doing the following is written:

1. Compile all test files using the original compiler.
2. Compile all test files using the new compiler (front end, analysis, and back end).

3. Compare the result of using the first compiler with the corresponding result of using the second compiler.

The batch file operates on a list of file names, making it easy to add additional test files.
After running the batch file, it is easy to inspect the result and see whether any errors were introduced in the new compiler and where these errors are. It is expected that some compiler errors will occur, as some of the test files test whether exceptions are thrown when they are supposed to. Errors, as shown in figure 2.7, are therefore accepted, as this error occur in both the original and the new compiler, and the subsequent error in the back end occurs because no output file was produced by the analytical part.



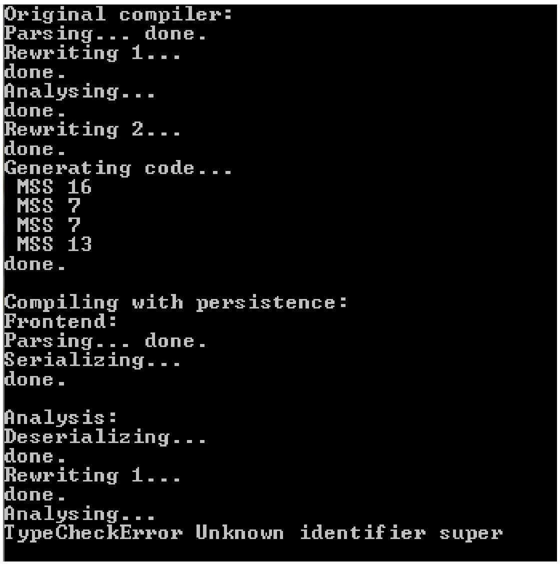
```
Testing for test/superclassAssign...

Original compiler:
Parsing... done.
Rewriting 1...
done.
Analysing...
TypeCheckError Types in assignment are not assignable (B,A)

Compiling with persistence:
Frontend:
Parsing... done.
Serializing...
done.

Analysis:
Deserializing...
done.
Rewriting 1...
done.
Analysing...
TypeCheckError Types in assignment are not assignable (B,A)

Backend:
Deserializing...
Generating code...
NullPointerException occurred. Xml-file may not have been analysed by the analys
e-part of the compiler
```

Figure 2.7: An example of an expected error.

2.5.2 Errors in the original compiler

During the testing several errors were encountered and fixed. Most of these resulted in changes in the implementation which is already covered in section 2.4. However, a few errors were encountered because of problems in the original compiler as discussed below.
In figure 2.8 the super-keyword does not work as it should. Instead, the keyword is treated as an identifier, which the compiler is unable to find, resulting in an exception being thrown. After using the debugger of eclipse and examining the code, the problem appeared to be the following code:

```
\footnotesize
```



```
Original compiler:
Parsing... done.
Rewriting 1...
done.
Analysing...
done.
Rewriting 2...
done.
Generating code...
 MSS 16
 MSS 7
 MSS 7
 MSS 13
done.

Compiling with persistence:
Frontend:
Parsing... done.
Serializing...
done.

Analysis:
Deserializing...
done.
Rewriting 1...
done.
Analysing...
TypeCheckError Unknown identifier super
```

Figure 2.8: An error encountered by using the super-operator in a test program to access content of the superclass.

```
if (this.name == "super") {
    if (IR.currentMethod.isStatic()) {
        throw new TypeCheckerException(
            "super encountered in static method."
        );
    }
    name = "this";
}
```

One of the errors is seen in figure 2.8. The problem here is that two `String`s are compared using the ==-operator. This operator tests equality for primitive types, but as `String` is an object, the operator instead checks whether the two strings refer to the same object, which is not intended. One may wonder why this has not been a problem before, but this is an example of the *string interning* used by the Java `String` class as described by the documentation for the class[19]. This implies that the `String`-class contains a pool of strings which is initially empty. When a string literal is initialized, the string pool is checked. If this contains a string equal to the one being initialized, the string from the pool is returned, resulting in two string literals having the same reference. This is the reason why the operator actually worked in the original compiler. However, if one of the strings are created by calling the constructor from the `String`-class directly, the

string will not be interned leading to the references being different, and the `==`-operator returning false when comparing the two strings, even though they are equal. This is what happens when strings are recreated in the deserializer. The code was corrected easily by using the `equals`-method instead, which checks for equal content and not equal references.
Another change made in the original compiler before circular references were able to be handled, and which is already mentioned in section 2.4, is the change of `Object`'s superclass to null and the addition of the `isTop`-field. This will not be changed back as the introduced solution is more intuitive, even though the serializer and deserializer are able to support this circular reference after being upgraded.

2.6 Discussion

The serializer and deserializer constructed in this project encountered many issues when being implemented, including loss of references, trouble with circular references, and accessing private fields. However, the result is a three-phase compiler where no serialization annotations are present and with a very readable and searchable IR between each stage. The alternative would be to use a serialization library, which would be much easier to apply, but would not be as flexible and as loosely coupled as the serializer created in this project. In a normal project, where development time equals development costs, creating a serializer from scratch would be expensive and a library would most likely be used. As the main purpose of the whole project is to provide a externally modular compiler for educational purposes and money was out of the question, the best solution was chosen regardless of the implementation time.
The current implementation of the serializer does not support arrays, as this is a special type of object which does not inherit the `Collection`-interface. This would be easy to implement in the same way collections were implemented, but no array is used anywhere in the compiler at this point. As no new fields are added by the users, this should not be a problem.
In regards to the design, treating the collections and maps different from other objects might not have been necessary. Serializing objects in general was not possible until the serializer was refactored to do so, which was the basis for treating collections and maps in another way. The serializer could be refactored to treat collections and maps like any other object, but the only advantage of doing this would be a cleaner serializer and deserializer. The disadvantages however would be an XML-file which would be much less readable and have a larger size. Consider a collection like `LinkedList`. This collection has a pointer to the first element, which has a pointer to the second element and so on. The tree structure for this would be much harder to read than the simple listing of elements currently used in this project. All fields of the map and collection objects would also be serialized, which include a lot of information that do not need to be passed on to the next module. In addition, support of arrays would need to be implemented before this would be possible, as many data structures implementing the `Map` and `Collection`-interfaces uses arrays in one way or another.
As a custom serializer was used in this project, a lot of time has been spent locating and correcting errors. The test files used to test the serializer covers most functionality, but there might be some special cases in which an error occurs due to an error in the serializer or deserializer, which it has not been tested for. This is always a difficult question; how much should software be tested before all errors are discovered?

As the tests covers most of the compiler features, more tests would most likely just be redundant. However, if additional tests are required, these are easily written and added to the test script as described in section 2.5.

2.7 Conclusion

This part of the project resulted in a compiler which works exactly the same way as the original compiler, but which is also externally modular. The compiler uses a custom serializer and deserializer which are both easy to expand, and minimal changes were introduced to the original compiler. The serializer and deserializer are completely separated from the compiler and both are only called by the `Compiler`-class once at most. The new compiler was also tested by using the test files for the original compiler. These were compiled using the original compiler, and then by using the new compiler. The files were then compared pairwise. After fixing the errors discovered by testing, all pairs of files ended up being equal, confirming that the compiler works as before for these test files.

Chapter 3

Part 2: Creating a Lego Mindstorm-module for the compiler

3.1 Introduction

This second part of the project is going to continue the work of the first part by providing a back end module for the compiler which can be used instead of the back end module of the first part. This back end will be able to generate executable code for the Lego Mindstorms EV3 robot.
In order to achieve this, the EV3 robot will be analysed and experimented on. The goal here is to execute a hello world program on the EV3. Once this is achieved, it is possible to design an API for the EV3 and use this and the original back end as basis for implementing the new module. Afterwards, the new EV3 compiler module will be tested, and the implementation and design will be discussed.

3.2 Analysis

This section covers the initial experimentation and analysis done on the EV3 brick and the associated software. The approach taken for assembling a hello world EV3 assembler program and executing this on the EV3 is also explained. Furthermore, the structure and data types of the EV3 assembler language will be explained.

3.2.1 Creating and running a bytecode program on the EV3

As a starting point, the EV3 should be able to run a simple hello world program. To do this, some way of transferring files to the EV3 is needed. This can be done by using a Secure Digital memory card (SD-card) or by using a cable from the computer to the device. As the computer used in this project does not contain a SD-card reader, the latter approach is used.
Although one might have expected the EV3-device to appear as an external storage device in Windows Explorer when connected to the computer, this is not the case. However, the system confirms by audio that a new device is registered, hence it should be possible to transfer files to the EV3.
For this reason, and for testing purposes, the software belonging to the EV3 is downloaded[1]. This software allows one to create programs for the EV3, using a drag-and-drop user interface. When running the program through the software, the program is transferred to the EV3 brick automatically and run. A file explorer for the EV3 is also included, where it is possible to download files from Windows Explorer to the EV3. It should also be possible to transfer files through bluetooth or Wifi according to the user guide[8], but during this project, only the file browser of the EV3-software will be used.
Initially, the brick is tested, using the EV3-software. A simple program is built

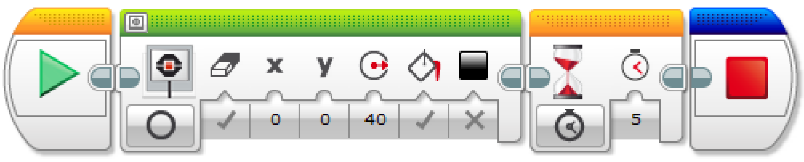


Figure 3.1: A program made using the EV3 software.

that draws a circle and waits five seconds as seen in figure 3.1. The software

[1]The software is free and can be downloaded at the Lego website[9].

compiles the program into a hexadecimal file with the extension .rbf, which is transferred to the EV3. Here, the program does as expected and the EV3 shows a filled circle which disappears after five seconds.
The next step is to create programs for the EV3 without the EV3 software. As the goal is to create a compiler which can compile MiniJava into assembler instructions which can be assembled into a runnable program, two options exist:

- Installing leJOS on the EV3 brick. leJOS is a Java virtual machine, which is able to run java byte code on the EV3.
- Finding documentation concerning byte code-instructions for the EV3, and either finding or creating an assembler.

Not a lot of experimenting has been done on the EV3 brick yet, as it is still very new, thus the leJOS virtual machine at this point is unstable. Furthermore, it requires a bootable SD-card and is difficult to install[10].
The second approach is more clear if byte code documentation can be found. After researching this, the firmware documentation[11] was found. This site contains all bytecodes and their corresponding hexadecimal-codes. It also include program examples which all state that a program header is needed. No documentation exists for the program header, or how this header is converted to hexadecimal, and as this is required, it is not possible to build an assembler from this. However, after examining the site more, a section about the lmsasm tool is found. This is an assembler which does the following (taken from the firmware documentation[12]):

1. Inserting the program and object headers
2. Translating the textual instructions to byte code values
3. Allocation of variables
4. Solving label jump addresses
5. Encoding of parametes in the byte code stream
6. Calculating sub call parameters
7. Generate an executable byte code file

This means that the compiler can compile into textual instructions, which the assembler can then convert into a .rbf-file which can be run at the EV3-device. The mentioned assembler is not found on the site however.
A git repository at [13] contains the source code for the EV3, including the

assembler mentioned above. After downloading the repository, the assembler is found in a stand-alone jar-file. The hello-world program found in the firmware documentation[14] is put into a text file with the file extension .lms, and the command below is run in a command prompt.

```
java -jar assembler.jar helloworld.lms
```

Initially, the assembler exits with an error, but after running the same command in admin mode a .rbf-file is created. Using the EV3-software, this is moved to the EV3-device, and executed. The result is seen in figure 3.2 where it is clear that the program is executed as expected.

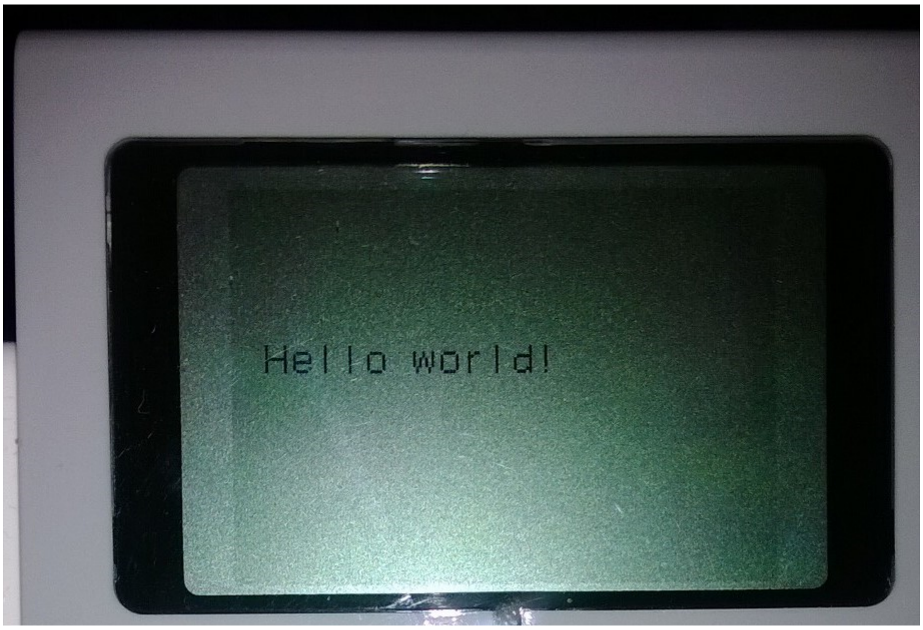


Figure 3.2: An assembled hello world byte code program running on the EV3

The assembler language assembled by the lmsasm tool is slightly different than the bytecode instruction set found at the firmware documentation site[11]. The assembler language is able to work with strings in a more straight forward way, while the corresponding byte codes require a more low level approach using arrays. After experimenting a bit, however, it is discovered that most of the instructions on the site can be used as a lmsasm-instruction if the prefix "op" is removed (e.g. the bytecode opADD32(DATA32, DATA32) corresponds to the lmsasm-instruction ADD32(DATA32, DATA32)).

3.2.2 The structure of an EV3 assembler program.

The structure of the EV3 assembler language is quite different and more high level than the byte code instructions used by the EV3. Each EV3 consists of at least one `vmthread` and zero or more `subcalls` which both contain instructions. The vmthreads are parallel running threads, making it fairly simple to implement threading for the EV3. The vmthread named main is the entry point for the application. Subcalls can take parameters and return values, making them ideal for methods.
No registers are used by the assembler language, nor is it possible to access the memory directly. Instead, a type strong variable system is used for memory management. The variable types available are all found on the website, but the types used in this project are these: `DATA8` `DATA16` `DATA32` `DATAS` Where the `DATA8`-type corresponds to a character or boolean flag, the `DATA16` is an integer and the `DATA32` is a long integer. The `DATAS` is a string data type, corresponds to an array of `DATA8` (a character array) and takes an integer as argument in order to specify the size of the containing string. The `DATA32`-variable will be used as the data type for a normal integer.
These variables can be used as global variables and as local variables in methods. Constants can also be used to define values. A sample program showing the program structure is found at the firmware documentation site[21] and has been included in figure E.1 in Appendix E for convenience.

Textual bytecode programs which can be assembled to runnable .rbf-files are now able to be written.

3.3 Design

In this section, the API which MiniJava will use for utilizing the special features of the EV3 will be designed and examined.

3.3.1 Creating a MiniJava API for the EV3

The EV3 include special features like the display, motors and sensors, that are not present on the LC3-processor that the compiler is currently generating code for. For utilizing these features, an application programming interface (API) need to be defined for the EV3.
The most intuitive would be to have a library with a set of static operations for controlling the IR. Using a command like the following for showing text would be very clean:
`EV3.displayText(int x, int y, String text)`
However, the compiler currently does not implement static methods in this way. Instead, an object has to be created first, where the static method can then be called from like seen below.

```
EV3 ev3 = new EV3();
ev3.displayText(int x, int y, String text)+.
```

An alternative would be to have some standard methods in the main class so that the signature would be `displayText(int x, int y, String text)`. However, this makes it impossible to group the methods in a library class, and may also conflict with methods created with the same name. The static instance method syntax shown above is therefore chosen for now.
Not all EV3 features will be implemented in this project. For now, the EV3 will have the following API:

`updateDisplay()`
Updates the display.
`clearDisplay()`
Clears the display.
`displayText(int x, int y, String text)`
Print the string `text` at the specified x and y coordinate. The `updateDisplay()` has to be called before the text is shown.
`wait(int milliseconds)`
Waits the specified amount of milliseconds before continuing execution.
`waitForButtonPress()`

Halts the execution until the middle button of the EV3 is pushed.

Motor control is not in the API at the time of this project being turned in, as it has not been possible to obtain a motor for testing. However, the API defined above should be enough for testing and demonstrating the new EV3 backend.
Adding new operations to the API is fairly easy. The full feature set of the EV3 can be seen by examining the byte codes found at the byte code documentation website[11]. Any feature that can be obtained by using a byte code or a combination of these can be added to the API. The implementation of new features and how to bind these to the API operations are covered in section 3.4.4.

3.4 Implementation

The new back end is now ready to be implemented. The implementation has been greatly inspired by the current LC3 back end implementation. The back end project made in the first part of the project is therefore copied, and the content of the `compiler.CODE` and `compiler.CODE.LC3` packages is deleted. The `compiler.CODE.LC3` is renamed to `compiler.CODE.EV3` and the content of all code generation methods are commented out of the program for now. The output file type is also changed to the .lms file extension. This results in a program skeleton for implementing the back end.
Not all features of the original compiler will be implemented in this project. Code generation for the object-oriented features of MiniJava has not been implemented, nor is it possible to create arrays or methods. Most other features have been implemented though, and the missing features are expected to be implemented in a special course following this project (see section 4.1).

3.4.1 The basis

To begin with, a helper class analogous to the `CODE`-class of the LC3 back end is created. This class is placed in the `compiler.CODE`-package. Two interfaces, `EV3` and `EV3Instruction`, are also created in the `compiler.CODE.EV3`-package. The latter class will become the superclass for all instruction classes added later. Both interfaces declare a `toString`-method that must be overwritten by the implementer classes.
A `Vector` of `EV3`-objects is inserted into the `CODE`-class. The `Vector`-data structure is similar to an array list, but also allows control over the initial capacity of the underlying array and the amount to increase the capacity with when the array is full. All elements in this vector will implement the `toString`-method defined in the `EV3`-class, which will return a string representation of the instruction. A vector of constants of type `EV3const` and two `EV3const`-fields are also added. These two constants will be assigned the values 1 and 0, and will be used for representing true and false.
Because of the structure of the assembler program that the back end should compile to, the only subclasses of the `EV3`-class is `EV3main` (The main vmthread), `EV3subcall` (subcalls), `EV3comment` (comments), `EV3variable` (global variables), and `EV3const` (constants), as these are the items allowed outside of vmthreads or subcalls. This means that the `EV3`-vector can only contain objects of these types. All instructions that are allowed inside vmthreads and subcalls are implementing the `EV3instruction`-interface. Note that some items are allowed both outside and inside a vmthread or subcall, for example variables (global and local), and implement both interfaces. Each subcall and the main vmthread

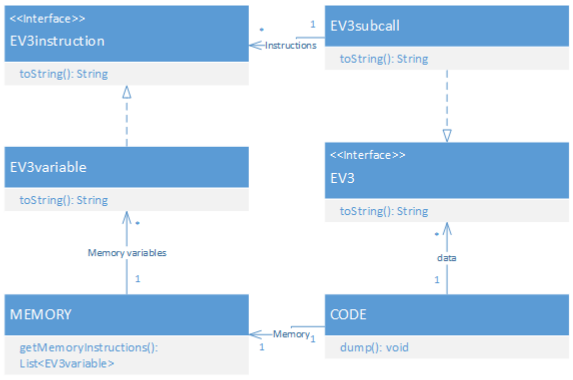


Figure 3.3: An overview of some important classes of the back end. Note that all machine instructions used inside methods are implementing the instruction-interface, and that only the methods involved in writing the code to an output file are shown in each class.

contain a vector of `EV3instructions` and of `EV3variables`. These two vectors contain the instruction set and the local variables of the operation respectively. The `CODE`-class also contains methods for commenting, a currentMethod field containing the current subcall or vmthread to be code generated, and the `dump()`-method which will create a `PrintStream` object, writing to the file with the output name defined in the main compiler class. The `toString`-method is first called on all constants and prints the result of these to the output file. A string with instructions is then obtained from the `MEMORY`-class explained in the next section, which is also printed to the file. The method finally calls the `toString()`-method on all items in the `EV3`-vector, including the subcalls and main vmthread. The `toString()`-methods in these classes will return a string representation of the entire operation by calling the implemented `toString()`-method on all `EV3instructions` of the vmthread or subcall. All instruction strings are then written to the output file. An overview of the classes involved is seen in figure 3.3.

3.4.2 Memory management

In the beginning stages of this back end, two global temporary variables were declared for each data type. When a statement like `a = 2 + 4` was evaluated, the 2 would be saved in the first `EV3DATA32`-variable and the 4 would be saved in the second. The plus operator would then apply the `ADD32` operation to the two temporary variables and save the result in the first temporary variable. This would work for most basic expressions, but for a more complex expression tree,

the number of temporary variables required would be unknown. A dynamic memory structure like the stack in the LC3-back end is needed for the temporary variables.
Unlike the LC3 assembler language which is converting all data types into integers, allowing for a generic stack, the EV3 language have more than one type, hence, more than one stack is needed. To keep track of temporary variables and which of these that are in use, the MEMORY-class is created. This class contains four array lists, one for each data type used in the assembler language. In addition, all lists have an index pointer, pointing to the top of the stack stored inside the array list. When a temporary variable to write to is requested, the appropriate push operation is called which then checks whether the appropriate index pointer equals the size of the array list containing the requested variable type. If it does, a new temporary variable is added to the array list and is then retrieved, and the index pointer is incremented. The retrieved variable can then be used to store data. If the index pointer is smaller than the size of the array, the index pointer points to a temporary variable which is already created but not currently in use. This variable is then returned.
If a temporary variable needs to be read from the stack, the index is first decremented. The temporary variable at that index is then returned. This means that all values saved in the temporary variables are meant to be stored once and read once. If the index pointer is a negative value after decrementing, an empty stack is being popped, and a CodeGenException is thrown.
One may ask why a stack data structure is not used for this instead of an array list. If a stack is used, the temporary variables saved in the list will disappear when popping the stack, resulting in those variables never being declared in the destination file. All variables that have been used in the stack need to be kept, so that they will still be declared when the dump-method is called in the end of the code generation phase.
As an example the expression 1+2*3, which is also used as an example in figure 2.2, is examined. A new temporary variable would first be created to store the number 1, followed by the back end storing it in the retrieved variable.
DATA32TMP[0] = 1.
Another temporary variable would then be created to store the number 2.
DATA32TMP[0] = 1
DATA32TMP[1] = 2
Then the right hand side of the multiplication is evaluated. DATA32TMP[0] = 1
DATA32TMP[1] = 2
DATA32TMP[2] = 3
The EV3DATA32-stack would then be popped twice, the values of the popped variables retrieved, and the multiplication instruction would be used to multiply the values. Then a temporary variable is reused by storing the result at the index pointer which will now have the value 1. DATA32TMP[0] = 1
DATA32TMP[1] = 6
DATA32TMP[2] = 3

Finally, two values are retrieved from the stack. These are added and the result is stored at the position indicated by the stack pointer, which will now be 0 as the stack is empty. `DATA32TMP[0] = 7`
`DATA32TMP[1] = 6`
`DATA32TMP[2] = 3`
The value 7 is now ready to be popped from the stack and used in a statement or another expression.

This implementation makes the back end able to translate any expression into EV3 assembler code without running out of temporary variables. The only disadvantage is that the memory allocated for evaluating expressions will be allocated for the rest of the program. In most cases, this amount will be small however. The alternative would be to use dynamic arrays in the assembler program to store these temporary variables. This would cost efficiency however, and is a bad option as the arrays would have to be constantly resized.

3.4.3 Code generation

All IR objects implementing a code generation method is traversed in the same way as described in section 2.2.2.3. When `MJMethod` is processed, the method name is checked. If this name equals the string "main", and the isEntry-field is set to true, this must be the main method. An `EV3main`-object is therefore created and added to the vector in `CODE`. The currentMethod of `CODE` is also set to be the `EV3main` object.
In the `MJBlock` all variables declared in the block are first added to the current method item. This is done by checking the types of the variables. If the type is boolean, a `DATA8`-variable is added to the variable list of the currentMethod-object. If the type is an integer, a `DATA32`-variable is added and if the type is a string, a `DATAS`-variable is added.
String variables are implemented similar to the memory system described in section 3.4.2. Each `DATAS`-object has two integer fields: A length-field and an inUse-field. The first one is the argument used to initialize the string when the `toString`-method is called. The second field is used to keep track on how much of this string variable which is currently in use. This allows the string variable to be reused without having to create a new `DATAS`-object by resizing the variable if needed.
If the variable is initialized when being declared, code is generated for the expression, which will leave the result on top of the stack corresponding to the type of the expression. As the source code is type checked in the analyse phase, the expression must be assignable to the variable. The temporary variable on top of the stack corresponding to the type of the variable is then retrieved, and a `MOVE`-command from the temporary variable to the declared variable is added.

When all variables have been added code is generated for the statements of the method. The code generation overview for these methods is seen below.

- `MJComment`: No code is generated for this.
- `MJAssign`: The method `getVariable()` in the `CODE`-class takes the name of a variable and returns the `MJvariable` with that name. It should always be possible to find a variable with that name, as the compiler in the analyse phase checked whether any variables not declared is being used. Code is then generated for the expression on the right hand side which is saved on top of the appropriate stack. The value from the temporary variable is then moved to the variable found.
- `MJIfElse`: The implementation of this statement is similar to the implementation found in the LC3-module. Code is generated for the condition expression first and then the temporary variable storing the result is supplied to the `EV3JR_FALSE`-instruction which will be added to the currentMethod-field. This instruction jumps to the specified label if the supplied variable is false. This label is placed after the this block. If an else block is supplied, a label will be placed at the end of this block. This makes it possible to skip the else-block if the condition is true by using a `EV3JR`-instruction which unconditionally jumps to the label.
- `MJWhile`: Similar to the `MJIfElse`. The difference is that the PC unconditionally jump back to the condition evaluation after the body of the loop have been executed. If the condition evaluates to false, the PC jumps past the while-statement, continuing execution.
- `MJMethodCallStmt`: Not fully implemented. Custom methods can not be used. However, the EV3 API defined in section 3.3.1 can be used as explained in section 3.4.4.
- `MJReturn`: Not implemented
- `MJPrint`: No reason to implement this, as the `displayText()`-method already exists for displaying text. The reason that this method is not reused for this is that no arguments for the position of the text is supplied. If these parameters should be added, the front end would have to be changed, which should be avoided.
- `MJPrintln`: Same as `MJPrint`.
- `MJBlock`: The `MJBlock` is already explained earlier in this section.

Below is an overview of which expression classes which have been implemented.

- MJBinaryOp: All subclasses of this class have a corresponding EV3instruction class. As described in the example in section 3.4.2 left and right hand side expressions are evaluated first. Two variables are then popped from the stack that corresponds to the type of the operators arguments and are processed using the appropriate instruction. The result is pushed onto the stack which contains variables having the same type as the result value of the expression. A MJLess-expression will therefore pop two values from the DATA32-stack and push the result onto the DATA8-stack. It is important to remember that if the left hand side is evaluated first, the first value popped from the stack will be the right hand side argument, and the second value popped will be the left hand side.
  A special case is the concatenation operator. Two string variables are retrieved from the EV3DATAS-stack. The inUse-fields of the two variables are added and the inUse-field of the destination variable is set to the sum in order to make sure that the destination variable has enough space for the concatenated string.

- MJUnaryOp: Not implemented.

- MJParentheses: Generates code for the content of the parentheses.

- MJIdentifier: First the EV3variable corresponding to the identifier is found, by calling the getVariable()-method in the CODE-class with the identifier name as the argument. After retrieving the variable, the data type of this variable is checked. An EV3MOVE-instruction is then added which will move the content of the retrieved variable to the temporary variable retrieved from the appropriate stack. In the case of an EV3DATAS-type, the inUse-field of the temporary variable has to be set to the value of the inUse-field of the variable retrieved, thereby making sure that the temporary string variable is big enough. When setting the inUse-field, the length of the EV3DATAS-variable is increased if inUse is greater than the current length.

- MJIfThenElseExpr: The same as the MJIfElse except the else is not optional.

- MJNew: Not implemented.

- MJMethodCallExpr: Not implemented.

- MJCast: Generates code for the identifier to be cast.

- MJNoExpression: No code generation is necessary here and the method is therefore empty.

- MJBoolean: Pushes the Const0-value on top of the EV3DATA8-stack if false or the Const1-value if true.

- `MJInteger`: Pushes the value of the integer on top of the `EV3DATA32`-stack.
- `MJNull`: Not implemented.
- `MJString`: First the quotes which the string is surrounded by are removed. Then a temporary `EV3DATAS`-variable is retrieved, and the inUse-field is set to be the length of the string. The string is the put on top of the `EV3DATAS`-stack.

3.4.4 The API

The compiler is now able to generate code for the EV3 for many of the features of the MiniJava language. The next step is to support the operations defined by the API in section 3.3.1.
Two main options exist for implementing the operations. The first is to change the front end and add new IR-classes for these operations, similar to the way the `System.out.print`-statement is handled in the original compiler. This would require a lot of work, and every time that a new operation is added the front end, analysis part and back end would all have to be changed quite a lot.
The second option is to add a isEV3Method boolean flag to the `MJMethod`-class. An EV3 MiniJava class would then be added to the list of classes in `MJProgram`, similar to the way the MiniJava String and Object classes are added. This class would contain all the EV3 operations, and have the isEV3Method-flag set. These methods are then able to be treated different from normal methods in the back end. In addition, implementing new operations is easy, as type checking and value initialization checks are already done for the `MJMethod`-class and method calls. The operations can be added to the EV3 MiniJava class, and the code generation class of `MJMethodCallStmt` or `MJMethodCallExpr` is then changed, depending on whether the method returns a value.
The analyse part of the compiler has to be changed in both options, as the type checking would halt the compiling process if these methods are not declared somewhere in the program. This seems fair as new functionality is added to the compiler, and not just a new target language. One may argue that the first approach would be more clean than the second one, as the `MJMethod` class is being used in two different ways in the second option. However, considering the advantages of doing so, the second option is chosen.
None of the operations of the API have return values, hence, the `MJMethodCall-Expr`-class does not need to be changed. It is therefore only necessary to alter the code generation of the `MJMethodCallStmt`-class. This is done by checking whether the target method bound in the analysis phase is an EV3Method. If it is, a switch-statement is used to switch between the code generation for the different operations. The implementations are seen below:

`clearDisplay()`:
An `EV3UI_CLEAR`-instruction is added to the instruction list, which clears the display of the EV3 by filling the whole display with the background color.

`updateDisplay()`:
An `EV3UI_UPDATE`-instruction is added.

`displayText(int x, int y, String text)`:
Code is generated for the second argument first. Then the first argument, and at last the last argument. This will leave the first argument on top of the `EV3DATA32`-stack followed by the second argument, and the string variable to display will be on top of the `EV3DATAS`-stack. An `EV3UI_TEXT`-instruction is then generated which instructs the display to draw the string at the specified x and y coordinate.

`wait(int milliseconds)`:
Code for the argument is generated first. In order to wait, a timer variable is needed in the assembler code. This timer variable is requested through the `getVariable()`-method in the `CODE`-class in case that it has already been added. If this variable has not been added yet, one is added and used along with the evaluated method argument for an `EV3TIMER_WAIT`-instruction. This instruction saves the value of the method argument into the timer variable and will start decrementing it. An `EV3TIMER_READY`-instruction with the timer variable as argument is also needed. This instruction will halt execution until the timer variable has the value 0.

`waitForButtonPress()`:
An `EV3UI_WAITFORBUTTON`-instruction is added, which halts execution until the button on the EV3 is pressed

Some problems with the EV3 types are encountered when implementing these methods. The `DATA32`-type is used for integers in the program. However, some of the assembler instructions take a `DATA16`-variable as an argument, hence the integer can not be supplied directly. A byte code for solving this is found however; the instruction `MOVE32_16`, which moves a `DATA32`-variable to a `DATA16` variable. This makes it possible to move variables from the `EV3DATA32`-stack to the `EV3DATA16`-stack. Before adding the instruction requiring `DATA16`-arguments, an `EV3MOVE`-instruction is added, which will move the `EV3DATA32`-arguments to the `EV3DATA16` stack, casting them in the process. The `EV3MOVE`-class has several constructors which will add the correct instruction depending on the data types supplied.

The implementation of the back end is now finished. As mentioned earlier, not all features are implemented yet

3.5 Test

Generating automated tests for the EV3 is not as easy as it was to generate tests for the first part of the project. No output files to compare with exists. Instead, the .lms files outputted by the compiler must be checked manually by inspecting these. If the program displays any output, it is also possible to test the program by assembling it and running it on the EV3. If any type errors or syntax errors exist, the assembler will not generate an output file.
If an effective and automated test method were to be constructed, a script merging all .lms-files into one could be written. All programs could then be executed sequentially with a delay between each program, making it easier to test the programs on the EV3. The MiniJava test files and the output files obtained by compiling are found in the attached archive (see appendix D).

- The DoesItWork.java file contains a program which uses most of the implemented features including string concatenation, if-statements, while-loops, the EV3 timer, the EV3 display, the EV3 wait-for-button command, the comparison and plus operators, identifiers and assignment. This test program can be run on the EV3 and works as expected.
- The Expressions.java file tests the memory system along with all binary operators by providing a long expression with many operators:
  `int result = (1*5+2*5*3-4) < 40 && (true == true && true);`
  As it is not possible to concatenate an integer to a string at this point, it is not possible to show the result on the EV3 display. However, the important result of this test is the number of temporary variables declared in the .lms file, which should be three `DATA8`-variables for evaluating the boolean parts of the expression, and three `DATA32`-variables for evaluating the integer parts. This matches the actual declarations seen in the .lms file.

The debugger in eclipse have also been used for locating errors when the compiler produced unexpected result code. In order to compile these files without having to run all three modules separately, a batch script was also written which executes all three modules sequentially, which is included in the attached archive (see appendix D.

3.6 Discussion

As mentioned earlier, not all features of the compiler have had code generation implemented for the EV3 in this project, nor have a lot of EV3 features been implemented. The project have been focusing on creating the most important parts of the new back end in a proper way. The memory management system implemented ensures that expressions can be evaluated without any problems, and the class hierarchy of the back end makes it easy to add new instructions and use these. It was never expected that the back end would be completed in this project, but this project has built a strong basis for finishing it.
In order to implement the EV3 back end, it was also necessary to add a new EV3 analysis module. Few changes were made in this module compared to the analysis part used by the LC3, but these changes were necessary. If the goal was to target a new assembler language and not add any new features to the compiler, no changes would be necessary to introduce to other parts of the compiler. However, as the EV3 has features that the LC3 does not support, these changes needed to be made in order to utilize these extra features.
The current API bind all operations to the EV3 MiniJava object. When more operations are added to the API it would make sense at some point to distribute these operations between more objects. This would mean a display library in a class `Display`, an `Audio`-class containing sound control and a `Motor`-class for motion control. This would be even more intuitive if static methods could be called directly from classes.

3.7 Conclusion

In this second part of the project, a new back end has been developed which is able to be substituted into the existing compiler and compile MiniJava source code into EV3 assembler code. It is then possible to assemble the assembler code by using an assembler, which will result in a .rbf file which can be executed on the EV3. A corresponding analysis module must be present too in order to use special features of the EV3 like the display and buttons. The back end module includes a memory management system similar to a stack which was developed in order to evaluate arbitrary large expressions without running out of temporary variables. This system, along with the rest of the implementation, makes it easy to complete the compiler, hopefully resulting in a very motivating and exciting tool for learning about compilers and robots.

Chapter 4

Conclusion

This project has studied and examined an existing compiler, and reviewed the options for making this compiler externally modular. A custom XML-serializer was built to write the intermediate representation of the program into a file, and a matching XML-deserializer was built in order to recreate the intermediate representation from the file. The result is a compiler consisting of three separate parts which are only connected through the file which is passed between the modules. With a few modifications, the serializer would also be able to be used as a general purpose loosely coupled XML-serializer if this was needed. For now, this part of the project resulted in a three-part compiler which will hopefully be a powerful tool for compiler education.
The compiler was then expanded with a back end for generating code for the Lego Mindstorm EV3 robot. Documentation for the assembler language was found on the internet, and the new back end was then implemented to convert the intermediate representation into the assembler language specified by the documentation. By using an assembler downloaded from the internet, this code was able to be assembled into a binary file and executed on the EV3. The fact that the compiled code is executable on the EV3 will hopefully be a strong motivating factor when this module is used for education.

4.1 Further work

This project will be continued in a special course in the spring of 2014 supervised by Associate Professor Christian Probst, where the new back end is expected to be completed. All current features of the compiler will be able to generate code to the EV3. This includes objects and fields, arrays, unary operators and method calls. The front end may also be enhanced to support static methods without having to create an object in order to allow for a more intuitive EV3 API.
The EV3 API will be extended so that more features of the EV3 are available, in particular motor and sensor control and perhaps audio. This would require more test programs which could also be used for introducing students to program the EV3 through MiniJava. Communication with the EV3 through Wifi might also be a topic of interest.
Enhancing the analyse phase could also be done, implementing assignment of the fields of the MiniJava String class and making use the public modifier.
The serializer could also be extended to support arrays and be made completely generic. This serializer could then be uploaded to the internet as a open source serialization library, hopefully helping other people lacking a loosely coupled lightweight Java library for serialization without the use of annotations.

Bibliography

[1] Roberts E. *An Overview of MiniJava*, **2001**, 5

[2] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns : Elements of reusable object-oriented software, chapter 5*, Addison-Wesley, 1995

[3] *Why is Java's reflection package considered the dark side of the force?*
17/2 2014
http://programmers.stackexchange.com/questions/170778/why-is-javas-reflection-package-considered-the-dark-side-of-the-force

[4] *Does JAXB use bytecode instrumentation? - Stack Overflow*
17/2 2014
http://stackoverflow.com/questions/2133732/does-jaxb-use-bytecode-instrumentation

[5] *Explaining the Document Type Definition*
17/2 2014
http://webdesign.about.com/od/dtds/a/document-type-definitions.htm

[6] *What is an XML Schema (XSD)*
17/2 2014
http://webdesign.about.com/od/schema/a/xml-schema-explained.htm

[7] *AccessibleObject (Java Platform SE 6)*
17/2 2014
http://docs.oracle.com/javase/6/docs/api/java/lang/reflect
/AccessibleObject.html#setAccessible(boolean)

[8] *User guide for the EV3*
17/2 2014 http://cache.lego.com/r/education/-/media/lego%20education/home/downloads/user%20guides/global/ev3/ev3-user-guide-en.pdf?l.r=-1623019953

[9] *LEGO.com Downloads*
17/2 2014
http://www.lego.com/en-us/mindstorms/downloads/software
/ddsoftwaredownload/download-software/

[10] *Developing with leJOS*
17/2 2014
http://sourceforge.net/p/lejos/wiki/Developing%20with%20leJOS/

[11] *Firmware Documentation*
17/2 2014
http://python-ev3.org/

[12] *Description of the lmsasm tool*
17/2 2014
http://python-ev3.org/description.html

[13] *LEGO MINDSTORMS EV3 source code*
17/2 2014
https://github.com/mindboards/ev3sources

[14] *Helloworld App (Windows PC)*
17/2 2014
http://python-ev3.org/helloworldwindows.html

[15] *Generics documentation - Java SE Documentation*
17/2 2014
http://docs.oracle.com/javase/7/docs/technotes/guides/language/generics.html

[16] *An introduction to C# Generics*
17/2 2014
http://msdn.microsoft.com/en-us/library/ms379564%28v=vs.80%29.aspx

[17] *Java hashmap key problems - Stack overflow*
17/2 2014
http://stackoverflow.com/questions/20201564/java-hashmap-key-problems

[18] *The Interface and Class Hierarchy Diagram of Java Collections*
17/2 2014
http://www.programcreek.com/2009/02/the-interface-and-class-hierarchy-for-collections/

[19] *Java String documentation (Java Platform SE 7)*
17/2 2014
http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern%28%29

[20] *EV3 variable types*
17/2 2014
http://python-ev3.org/variabletypes.html

[21] *EV3 program structure*
17/2 2014
http://python-ev3.org/programstructure.html

Appendix A

Notation

The notation used for the context free grammars defined in this project is given below.

- A production for a variable has the syntax shown below. Literals are enclosed in quotation marks and references to other variables can be used. Spaces between items mean that any number of spaces, tabs, or new lines are allowed.
  ```
  <production-name>  ::= "literal" <another-production>
  ```
- A variable can have several productions by using the or-operator.
  ```
  <production-name> ::= "production one"
                      | "production two"
  ```
- The closure (*) operator can be used to express zero or more occurrences of a variable.
  ```
  <production-name> ::= <repeated-variable>*
  ```
- Parentheses are used in order to group variables and literals.
  ```
  <production-name> ::= (<repeated-variable> "repeatedliteral")*
  ```
- Parentheses can also be used with the or-operator.
  ```
  <production-name> ::= ( "this" | "or this") "but definitely this"
  ```

- Square brackets are used to express that the item is optional.
  ```
  <production-name> ::= ["maybe this"] "but definitely this"
  ```
  This is equivalent to
  ```
  <production-name> ::= "maybe this" "but definitely this"
                      | "but definitely this"
  ```

The notation for the regular expressions used in this project is given below.

- A range of values is defined between two parentheses.
  `('f'..'h')` (the letters from f to h).
- The ? operator means zero or one occurence of a character.
  `'a'? 'b'` (either ab or b).
- The or operator and closure operator are allowed in regular expressions as well.
  `('a' | 'b')*` (any combination of a's and b's)
- The + operator is used as a "one-or-more"-operator.
  `('a' | 'b')+` (Any non-empty combination of a's and b's.)

Appendix B

Grammars

```
<object>       ::= "<" <CLASSNAME> " id=" <INTEGER> ">"
                   (<field>*|<ALREADYSERIALIZED>) "</" <CLASSNAME> ">"
               |   "<" <CLASSNAME> " id=" <INTEGER> ">" (<field>*|<ALREADYSERIALIZED>)
                   "</" <CLASSNAME> ">" null "</" <CLASSNAME> ">"
<field>        ::= "<" <FIELDNAME> ">" <fieldcontent> "</" <FIELDNAME> ">"
<fieldcontent> ::= <object> | <primitive> | <collection> | <map>
<primitive>    ::= "<" <PRIMITIVENAME> ">" <PRIMITIVEDATA> "</" <PRIMITIVENAME ">"
<collection>   ::= "<" <COLLECTIONCLASS> " id=" <INTEGER> ">"
                   (<fieldcontent>*|<ALREADYSERIALIZED>) "</" <COLLECTIONCLASS> ">"
<map>          ::= "<" <MAPCLASS> " id=" <INTEGER> ">"
                   (<mapelement>*|<ALREADYSERIALIZED>) "</" <MAPCLASS> ">"
<mapelement>   ::= "<KeyValuePair>" <mapkey> <mapvalue> "</KeyValuePair>"
<mapkey>       ::= "<key>" <fieldcontent> "</key>"
<mapvalue>     ::= "<value>" <fieldcontent> "</value>"
```

Figure B.1: The BNF for the formal syntax of the serialization format.

```
<program>            ::= <main-class> <class>*
<main-class>         ::= "class" <IDENTIFIER> "{"
                              "public" "static" "void" <IDENTIFIER> "(" "String[]"
                              <IDENTIFIER> ")" <block>
<class>              ::= "class" <IDENTIFIER> ["extends" <IDENTIFIER>] "{"
                              <var-declaration>* <constructor-declaration>* <method-declaration>*
<block>              ::= "{" <var-declaration>* <statement>* "}"
<var-declaration>     ::= <type> <IDENTIFIER> ["=" <expression>] ";"
<type>               ::= "boolean" | "int" [ "[]" ] | <IDENTIFIER>
<method>             ::= ["public"] ["static"] <return-type> <IDENTIFIER> "("
                              [<type> <IDENTIFIER> ("," <type> <IDENTIFIER>)* ] ")" "{"
                              <var-declaration>* <statement>* "return" [<expression>] ";" "}"
<constructor>        ::= ["public"] ["static"] <IDENTIFIER> "(" [<type> <IDENTIFIER>
                              ("," <type> <IDENTIFIER>)* ] ")" "{"
                              <var-declaration>* [<constructor-call>]
                              <statement>* "return" ";" "}"
<return-type>        ::= <type> | "void"
<constructor-call> ::= ("super" | "this") "(" [<expression> ("," <expression>)* ] ")" ";"
<statement>          ::= <COMMENT>
                       | <block> |
                       | "if" "(" <expression> ")" <block> ["else" <block>]
                       | "while" "(" <expression> ")" <block>
                       | <id> "=" <expression> ";"
                       | "System.out.println" "(" <expression> ")"
                       | "System.out.print" "(" <expression> ")"
                       | <id> "(" [<expression> ("," <expression>)*] ")" ";"
<expression>         ::= <level-1> ("&&" <level-1>)*
<level-1>            ::= <level-2> ("==" <level-2>)*
<level-2>            ::= <level-3> ("<" <level-3>)*
<level-3>            ::= <level-4> (("+" | "-") <level-4>)*
<level-4>            ::= <level-5> ("*" <level-5>)*
<level-5>            ::= "-" <level-5>
                       | "!" <level-5>
                       | "new" "int" "[" <expression> "]"
                       | "new" <IDENTIFIER> "(" [<expression> ("," <expression>)*] ")"
                       | <id>
                       | <id> "[" <expression> "]"
                       | <id> "(" [<expression> ("," <expression>)* ] ")"
                       | "(" <expression> ["?" <expression> ":" <expression>] ")"
                       | "(" <IDENTIFIER> ")" <id>
                       | "true"
                       | "false"
                       | "null"
                       | <INTEGER>
                       | <STRING>
<id>                 ::= <IDENTIFIER> | <this-id> "." <IDENTIFIER>
<this-id>            ::= "this" | "super" | <IDENTIFIER>
```

Figure B.2: The BNF grammar for MiniJava used in the project.

Appendix C

Diagrams

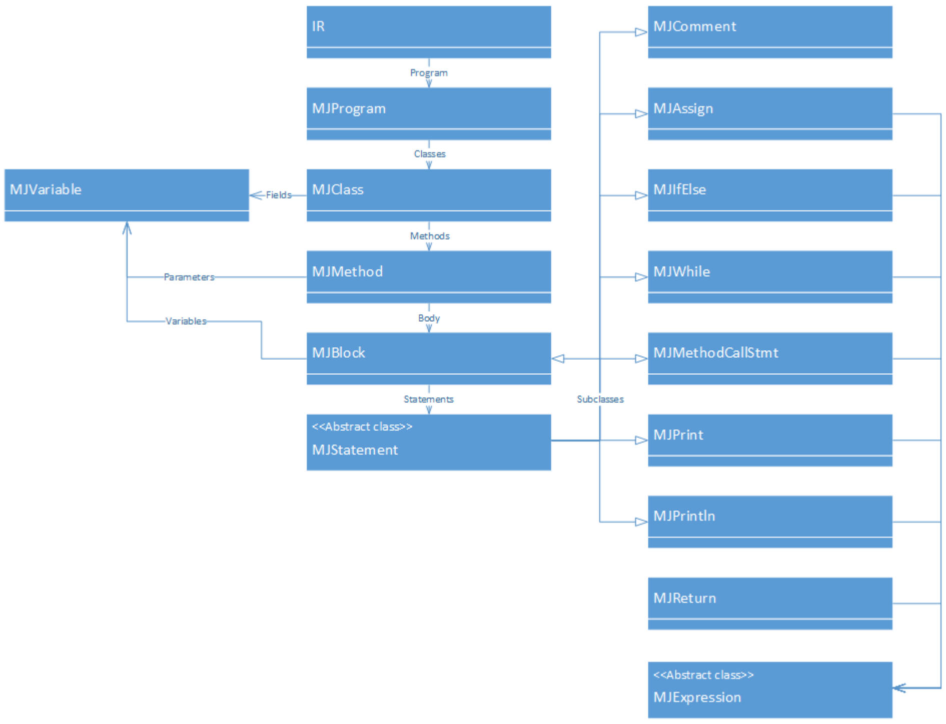


Figure C.1: The first part of the overview of the original compiler.

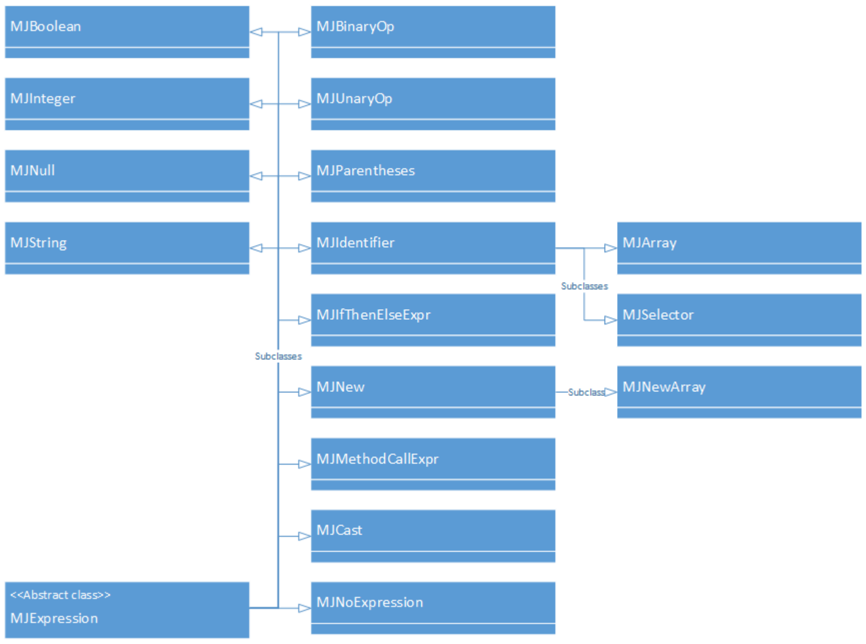


Figure C.2: The second part of the overview of the compiler. The class hierarchy of the expression class is seen here.

Appendix D

The attached archive

A zip-archive is handed in with this project and contains the following directories and files:

- Part 1
  - Jar files
    - FileEqualityChecker.jar
      The program used for testing equality of files. Invoked with two file paths.
    - MiniJavaCompiler_Analysis.jar
      The analysis part of the compiler.
    - MiniJavaCompiler_Backend.jar
      The back end part of the compiler.
    - MiniJavaCompiler_Frontend.jar
      The front end of the compiler.
    - MiniJavaCompiler_Original.jar
      The original compiler before any enhancing was done.
  - Source
    - The sources of the jar files mentioned under Jar files. These can be imported into an eclipse project.

  - Test
    - Test files along with the test script and all necessary jar files to run the test script.
- Part 2
  - Assembler
    - The lmsasm-tool described in the second part of the project. This tool assembles .lms files into runnable .rbf files. This tool has to be run from the command line in admin mode.
  - Jar files
    - MiniJavaCompiler_MindstormAnalysis.jar
      The new EV3 analysis module.
    - MiniJavaCompiler_MindstormBackend.jar
      The new EV3 back end module.
    - MiniJavaCompiler_Frontend.jar
      The front end of the compiler, the same as in part 1.
  - Source
    - Sources of the jar files in the Jar files directory. These can be imported into an eclipse project.
  - Test
    - Test programs and the compile batch script. This directory also contains the jar files necessary to run the compile script along with the .xml and .lms files produced by running the script.

Appendix E

Programs

```
                                                   // Constants are declared in between
                                                   // objects (vmthread and subcall)
define    MY_GLOBAL_CONSTANT           100         // MY_GLOBAL_CONSTANT equals 100

                                                   // Global variables are declared
                                                   // in between objects
DATAF     MyGlobalFloat                            // MyGlobalFloat is of the type float


vmthread  MAIN                                     // All programs must start with the
                                                   // "main" thread object
                                                   // (more vmthread's are allowed)
{
                                                   // Local variables are declared
                                                   // inside objects and are only valid here
  DATA8   MyLocalByte                              // MyLocalByte is of the type signed byte
Loop:                                              // Labels are local and only recognised
                                                   // inside the object (Symbolic names
                                                   // can be reused in other objects)

  CALL(MySubcall,MyLocalByte)                      // Call MySubcall with one parameter

  MOVE8_F(MyLocalByte,MyGlobalFloat)               // Assign the return parameter value
                                                   // from MySubcall to MyGlobalFloat

  JR(Loop)                                         // Jump unconditional to label "Loop"
}

subcall   MySubcall                                // Sub calls are all global objects
{
  IO_8    MyParameter                              // Declare sub call parameters

  DATA8   MyLocalVariable                          // Declare sub call local variables

  MOVE8_8(MY_GLOBAL_CONSTANT,MyLocalVariable)      // Initialise MyLocalVariable

  MOVE8_8(MyLocalVariable,MyParameter)             // Return the value of MyLocalVariable
                                                   // in parameter MyParameter
}
```

Figure E.1: The program structure of an EV3 assembler program.[21]