LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF THEORETICAL COMPUTER SCIENCE AND THEOREM PROVING

Implementation of a compiler for MiniJava

Lemar Abawi

Bachelor's Thesis
Computer Science

Supervisor: Prof. Dr. Jasmin Blanchette

Advisor: Lydia Kondylidou

Submission Date: October 31, 2024

Abstract

This thesis presents the development of a compiler for MiniJava, a simplified subset of the Java programming language. The compiler is designed to generate Java bytecode, which can be executed by the Java Virtual Machine. All standard compiler phases are described and an optmisation technique is presented. ATNLR, a parsing software, is used to aid with the syntactic analysis and Jasmin, a Java bytecode assembler, handles the bytecode generation.

Contents

1 Introduction

1.1 Programming languages

Programming languages are used to translate algorithmic logic into instructions that a computer can process. Usually, one diferentiates between high-level languages (e.g., Python, Java) that abstract away hardware details, providing user-friendly syntax. Low-level languages (e.g. Assembly, Machine code) are closer to the hardware, allowing fine-grained control.

Another split of programming languages can be made between compiled and interpreted languages: Compiled languages (e.g., C, Rust) are translated into machine code before execution, enhancing runtime performance by allowing the system to run optimized, low-level instructions directly. In contrast, interpreted languages (e.g., Python, Ruby) execute code line-by-line through an interpreter at runtime, prioritizing ease of use over raw execution speed. This differentiation is heavily tied into statical and dynamical typing. While the former enforces objects to be of a certain subtype at runtime, the latter does not and thus enables a more flexible approach to programming.

Since programming is predominantly done using high-level languages, there exists a need for a tool to translate these abstractions into machine code, the language understood by the computer. Those are called compilers.

1.2 A brief history on compilers

Machine code is the representation in binary (or a variation thereof, i.e. hexadecimal) of a computer program that the CPU can execute immediately. As computers were not mass produced in the first half of the 20th century, there was no need to have unified hardware architectures, which lead to non-reusable machine code. Since machine code instructions usually entail several intricacies like evaluating arithmetic expressions, saving and loading information to and from memory and jumping between instructions, this quickly becomes hardly maintainable for the human mind. The need for a more abstract representation prompted the invention of some of the first compiler in the early 50s (1).

This birthed the formulation of the very first languages considered high-level, like `FORTRAN` and `COBOL`, which were accompanied by a compiler translating the higher-level syntax to machine code. The evolutionary "Portable C Compiler", that came with the C programming language in the 70s, introduced as one of the first the idea of unifying the translation process, thereby making code reusable across hardware architectures (2). Sun Microsystems set the gold standard for this idea two decades later by creating a virtual machine (`JVM`) for their lang´guage Java, having a "Just-In-Time" compiler, that takes bytecode generated by the java compiler itself and translates it to machine code during execution on the target architecture, as one of its most prominent features used for optimisation (3). This allowed for even better portability and made Java an industry defining language.

Today, programming is more accessible than ever thanks to very advanced compilers that not only serve as translation tool, but throw warnings and errors, if code is written unconventionally or erroneously and optimize code output.

1.3 Objective

This thesis documents the development of a compiler for the langage `MiniJava`, a subset of Java. The compiler shall generate Java bytecode, which can be passed on to the JVM for execution. This approach is shared by many Java-like language (e.g. Kotlin, Groovy, Scala).

1.4 Compiler phases

Languages targeting the JVM follow the standard phases of compilation:

1. **Lexical Analysis:** The source code written in a JVM-targeting language is scanned, and its character sequences are converted into so called tokens.
2. **Syntax Analysis:** The tokens are parsed according to the language's grammar rules to generate a syntax tree, representing the program's structure.
3. **Semantic Analysis:** The compiler checks for type- and scope correctness, and further semantic rules that cannot or barely be expressed by grammar restrictions.
4. **Java Bytecode Generation:** The JVM understands the Java instruction set, which is then translated into the correct hardware instruction set.

1.5 Grammars and yyntax Trees

Formal grammars are an object of interest across computer science: A formal grammar G is defined as a quadruple $G = (N, \Sigma, P, S)$, where:

1. **Alphabet** Σ: A finite set of symbols known as terminals. These symbols are the indivisible elements used to construct strings in the language.
2. **Non-terminals** (N): A finite set of symbols that represent patterns of terminals. These symbols are placeholders that can be replaced according to the production rules.
3. **Production rules** (P): A finite set of rules that specify how non-terminal symbols can be replaced by sequences of non-terminals and terminals. Each rule is of the form:

   $$A \to \alpha$$

   where $A \in N$ (a non-terminal) and α is a string consisting of symbols from N and Σ (i.e., $\alpha \in (N \cup \Sigma)^*$).
4. **Start symbol** (S): A special non-terminal symbol from N that serves as the starting point for generating strings in the language.

A string s is considered to belong to the language $L(G)$ generated by grammar G if there exists a derivation from the start symbol S to s composed exclusively of terminal symbols, denoted as:

$$S \Rightarrow^* s$$

where $\Rightarrow^*$ indicates that s can be derived in zero or more steps.

Context-free grammars (CFGs) impose additional restrictions on the definition of formal grammars. A context-free grammar is defined as a formal grammar where all production rules adhere to the following form:

$$A \to \alpha$$

where $A \in N$ is a single non-terminal and α can be any string of terminals and non-terminals (i.e., $\alpha \in (N \cup \Sigma)^*$).

Since these restrictions mean that nested structures can be represented well by recursion, while also being able to profit from the well established and efficient parsing algorithms specifically created for CFG-languages, programming languages are canonically presented in this form.

Every CFG derivation can be represented as a *syntax tree* by making every non-terminal a node and every terminal a leaf. In the context of compiler construction, leaves are called tokens. An exemplary assignment operation in MiniJava, specified as a syntax tree, could look like this:

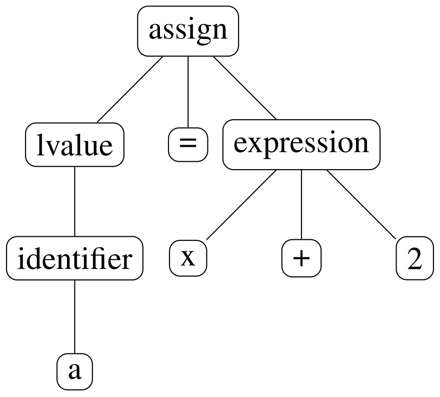

Figure 1: Syntax tree representation of an assignment expression

1.6 Regular expressions

Regular expressions are an important concept to simplify the formulation of the grammar of the compiler. Formally, a regular expression over an alphabet Σ is defined inductively:

1. $\emptyset$ is a regular expression and represents the empty set.
2. ε is a regular expression and represents the language containing only the empty string.
3. For any $a \in \Sigma$, a is a regular expression representing the language $\{a\}$.
4. If r_1 and r_2 are regular expressions, then:
    - $r_1 r_2$ is a regular expression representing the concatenation of the languages r_1, r_2.
    - $r_1 \cup r_2$ is a regular expression representing the union of the languages r_1, r_2.
    - r_1^* is a regular expression representing the Kleene star of r_1, which denotes zero or more repetitions of strings in the language defined by r_1.

Both regular expressions and formal grammars are important theoretical concepts that later will be applied in the parsing stage of the compiler.

1.7 Handwritten vs. generated parser

The obvious drawback of a handwritten parser is the much bigger implementation effort added to a compiler project. It leaves, however, the developer in complete control of the project contents. This is especially important when considering to implement different optimizations alongside the canonical compiler phases, since that requires a high degree of flexibility and understanding of the internals.

Generating the parser means the developer can declare their CFG comfortably and trust someone else's expertise in the area to produce a correct parser.

The first parser generators emerged in the late 1960s, including tools like `YACC` (Yet Another Compiler Compiler), developed in 1970 at Bell Labs. Yacc used algorithmic parsing techniques to generate parsers from CFGs, which has been improved upon dramatically in the following years culminating in the 90s with the introduction of ANTLR.

1.8 ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for handling structured text or binary files (4). Its first version was released in 1992 as part of the Purdue Compiler Construction Tool Set. Since then, ANTLR has evolved enormously up to its current fourth version introduced in 2013. The user provides a grammar, through which ANTLR generates a lexer, parser and optionally visitor interface, that is used to traverse the syntax tree.

1.9 MinJava

MiniJava is, as the name suggests, a simplified subset of the Java programming language. It specifically excludes certain advanced Java features, providing a more approachable experience for learners. Notable features that MiniJava keeps:

- loop and control flow capability
- classes, methods, fields
- arrays

It purposefully omits complex concepts like polymorphism, inheritance, and generics and restricts itself to `int`, `bool`, custom classes and their respective array implementation as data types. By doing so, MiniJava limits its scope, focusing primarily on learning how basic programming works.

As a stripped-down version of Java, MiniJava can be thought of as an educational object-oriented programming language. This makes it an excellent tool for teaching the fundamentals of OOP without overwhelming learners, especially those new to programming or those at earlier educational stages, such as school students. To get an exact understanding, please refer to the appendix for the formal definition.

2 Implementation

Java is chosen for the implementation due to its seamless integration and setup with both ANTLR and Jasmin, an assembler for Java bytecode used at the bytecode generation stage.

2.1 Parsing

The Java classes needed for parsing are generated by ANTLR: The `CharStream` reads raw text and provides the lexer `GrammarLexer` with the characters one by one. Once the lexer reads those characters and generates tokens, the `CommonTokenStream` collects and passes the tokens to `GrammarParser` which checks the sequence of tokens for grammar adherence and constructs a syntax tree:

Main.java

```
...
CharStream input = CharStreams.fromFileName(fileName);
GrammarLexer lexer = new GrammarLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
GrammarParser parser = new GrammarParser(tokens);

ParseTree tree = parser.program();

MyVisitor visitor = new MyVisitor();
visitor.visit(tree);
...
```

The grammar classes refer to a grammar file, making use of a mix of CFG rules described by regular expressions, the user created to declare his intended language: All tokens are defined using a syntax extending the basic regular expression definition above. Certain non-terminals can also benefit from this syntax, because rules that would typically involve recursion in the context of CFG can be expressed more concisely with the use of the Kleene star operator.

Grammar.g4

```
...
method_arguments: expression (',' expression)*;
...
IDENT: [a-zA-Z_][a-zA-Z_0-9]*;
NUMBER: [0-9]+;
...
```

2.2 Visitor pattern

To fully utilize the structure of the syntax tree, it becomes essential to introduce a design pattern that enables systematic traversal of the tree's nodes. The program must assess each node, performing operations such as scope - and type checking. The design pattern that perfectly fits this requirement is the visitor pattern, which provides a method of evaluating each node without altering its structure.

By leveraging this pattern, a program can perform specific operations based on the node type while maintaining a clean separation between the tree's data structure and the logic needed for analysis. Applied onto MiniJava compiler context, this is the high-level approach:

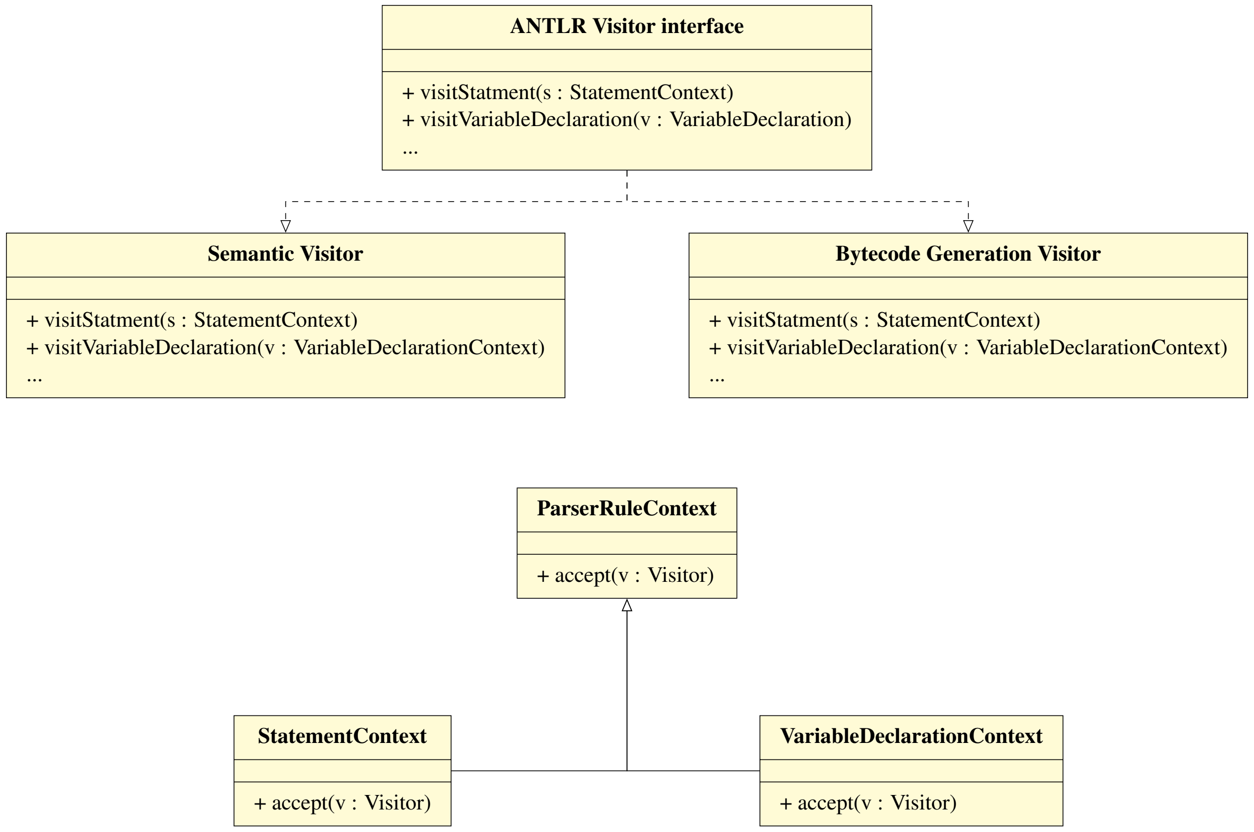
ANTLR Visitor interface
+ visitStatment(s : StatementContext)
+ visitVariableDeclaration(v : VariableDeclaration)
...
Semantic Visitor
+ visitStatment(s : StatementContext)
+ visitVariableDeclaration(v : VariableDeclarationContext)
...
Bytecode Generation Visitor
+ visitStatment(s : StatementContext)
+ visitVariableDeclaration(v : VariableDeclarationContext)
...
ParserRuleContext
+ accept(v : Visitor)
StatementContext
+ accept(v : Visitor)
VariableDeclarationContext
+ accept(v : Visitor)

Figure 2: UML diagram of the Visitor pattern

2.3 Syntax tree and symbols

The syntax tree generated by ANTLR is a linked tree data structure composed of `ParserRuleContext` instances as non terminal nodes and `TerminalNode` instances as leaf nodes, which will be referred to as symbols for simplicity in the following discussion. By default, this tree is traversed in a depth-first manner, starting from the leftmost node at each level, by recursively invoking the predefined `visitChildren()` method on each node. However, visiting every node is unnecessary, as many (such as leaf nodes) do not require processing. To assess specific nodes, all necessary `visit()` methods must be overridden in a custom visitor class, as ANTLR's default implementations do not include any specific assessments.

2.4 Scope checking

One of the primary responsibilities of compilers is to prevent runtime errors, as machine code error messages are often difficult to understand or, in some cases, entirely absent, although the program has terminated prematurely. To achieve this, programmers do rely heavily on scope- and type checking.

Scopes are abstract representations of hierarchy within the code. They remove any ambiguities regarding the use of symbols:

- Every time a bracketed statement is opened, a new scope is created
- Symbols declared in outer scopes remain accessible within inner scopes; however, they can be shadowed by redeclaration within an inner scope.
- Additionally, an identifier cannot be associated with multiple symbols simultaneously, and the use of variables within expressions implies that they have been declared (and initialized) in an outer scope.

To manage this process, a data structure called the `SymbolTableStack` is introduced.

2.4.1 Symbol Table

A `SymbolTable` is essentially a map `Map<String, Symbol>` linking strings to symbols. Whenever a variable is declared in the current scope, the compiler checks whether the identifier already exists in the current symbol table. If it does not, the identifier is stored in the `SymbolTable` instance, associating it with the corresponding syntax tree node. If the identifier is already present, an exception is raised:

SymbolTable.java

```java
public class SymbolTable {
    private Map<String, Symbol> symbols;

    public SymbolTable() {
        this.symbols = new HashMap<>();
    }

    public void add(Symbol symbol) {
        if (symbols.get(symbol.getName()) != null)
            ExceptionUtils.throwException(...);

        symbols.put(symbol.getName(), symbol);
    }
}
```

2.4.2 Stack of Symbol tables

As the name suggests, the `SymbolTableStack` is a stack composed of `SymbolTable` elements. If one descends a bracket layer, a new `SymbolTable` instance is pushed onto the stack and, once the visit of the body of the inner layer is finished, popped off again.

MyVisitor.java
```
...
@Override
public Object visitClass_declaration(Class_declarationContext ctx) {

    // Add the class symbol to the current symbol table
    tables.peek().add(ctx);
    tables.push(new SymbolTable());

    // Visit inner layers
    visitChildren(ctx);

    // Pop the current symbol table of the stack
    tables.pop();
    return null;
}
...
```

This ensures access to all variables declared in outer scopes while preventing access to variables declared in inner scopes.

2.4.3 Lookups

The `SymbolTableStack` can now be used to enforce the three restrictions described earlier that must apply to a proper scoping of MiniJava. There are four types of symbols must be checked for their scoping:

- Local variables
- Classes
- Fields
- Methods

Lookups are made during the whole traversal of the compilation, since more information about the used symbols are needed all the time.

For each of the four symbols a slightly different traversal of the stack is needed. In the case of variables, the traversal checks through all locally available variables until the desired variable declaration is found or returns `null`, which then is used to raise an exception in the visitor.

SymbolTableStack.java

```
...
public static Variable_declarationContext lookupVar(String name) {

    // Traverse the stack from top level to main class level
    for (int i = symbolTableStack.size() - 1; i >= 2; i--) {
        SymbolTable currentTable = symbolTableStack.get(i);
        Symbol symbol = currentTable.get(name);
        if (symbol != null) {
        // Return the succesfully retrieved symbol declaration
            return (Variable_declarationContext) symbol;
        }
    }
    return null;
}
...
```

The complexity of accessing variables across scopes is $O(n)$, where n represents the number of scopes. However, in most cases, variable declarations are typically made within the same scope or one layer up, which effectively reduces the access time to near constant. This common structure simplifies variable lookup and minimizes the performance impact, as variables can often be resolved without traversing multiple layers.

2.5 Type checking

Type checking is the second important stage to prevent runtime errors. This task can be split into two related parts:

- Every expression must be resolvable to exactly one type (i.e. `true && 3` cannot be tolerated)
- The resolved type must match the type that is expected in the context of the expression (i.e. `i = 2;` cannot be tolerated if `i` has been declared as a `boolean`)

By checking that the actual type can be resolved and matches the expected type, the JVM will not crash due to invalid memory access or similar. Since an expression has zero, one or two subexpressions, it seems reasonable to typecheck recursively:

TypeChecker.java

```java
public static void typeCheck(ExpressionContext ctx, Object expected) {
    switch (ctx.expression().size()) {
        case 0:
            checkConstantExpression(ctx, expected);
            break;
        case 1:
            checkUnaryExpression(ctx, expected);
            break;
        case 2:
            checkBinaryExpression(ctx, expected);
            break;
    }
}
```

The subroutines evaluate the expression against the expected type and raise an error if the type is not resolvable or does not match the expected type. If there is at least one other subexpression, `typeCheck(...)` is called recursively. While the expected type of the whole expression must be determined by the context, the expected type for further subexpressions are determined by the kind of operator being used. Expressions that are expected to be non primitive can be resolved immediately, since they can only appear as constants either via instantiation or reference.

Types are divided into primitive types (such as `int`, `boolean`, and their respective array implementations) and non-primitive types (which include classes and their array implementation). The former are represented as `int`, the latter as `String` requiring the common parent class `Object` to be used as argument type during typechecking.

TypeChecker.java

```
public static void checkUnaryExpression(ExpressionContext ctx,
Object expectedType) {
    if (ctx.MINUS() != null) {
        Object actual = SimpleType.INT;
        if (!actual.equals(expectedType))
            ExceptionUtils.throwException(...);

        typeCheck(ctx.expression(0), SimpleType.INT);
        return;
    }

    if (ctx.LPAREN() != null && ctx.RPAREN() != null) {
        typeCheck(ctx.expression(0), expectedType);
        return;
    }

    if (ctx.NOT() != null) {
        Object actual = SimpleType.BOOL;
        if (!actual.equals(expectedType))
            ExceptionUtils.throwException(...);

        typeCheck(ctx.expression(0), SimpleType.BOOL);
        return;
    }
    ExceptionUtils.throwException(...);
}
```

Since the grammar and the used types are simple, there is no need to establish the typechecking formally via lambda calculus or similar.

2.6 Exception handling

One of the key objectives a compiler must fulfill is to clearly report what went wrong during compilation. Broadly, three kinds of errors can be identified:

- **Syntax errors**: The code does not conform to the grammar rules.
- **Scope errors**: A symbol is either used without declaration or declared more than once within the same scope.
- **Type errors**: The expected and actual types of an expression do not match.

The first category is managed by `ANTLR`, while for the latter two, simple error messages are output to the console including error line and column and the cause :

ExceptionUtils.java

```
...
public static void throwException(int line, int column,
String key, Object... params) {
    String errorMessage = getMessage(key, params);
    throw new CompilerException(line, column, errorMessage);
}
...
```

2.7 Bytecode generation

After syntactic and semantic correctness have been established, the Java bytecode is generated: With the help of the the Java bytecode assembler Jasmin, the process of bytecode generation is simplified heavily by offering a human-readable assembly language that directly maps to Java bytecode.

2.7.1 The Java Virtual Machine

The Java Virtual Machine (JVM) is an interpreter that enables a computer to run Java programs and programs written in other languages that are compiled to Java bytecode. The JVM starts by loading .class files, which contain the compiled bytecode. The class loader manages the loading of classes and ensures that they are loaded only once during the execution of a program. It also handles the linking of classes, including verification, preparation, and resolution of references. Any Java class file is made up of several segments (5):

- **Base data:**
  - References the class's name, its superclass, and its implemented interfaces.
  - Contains metadata such as the Java version number, annotations, and the name of the source file used for compilation, also called attributes.
- **Constant pool:**
  - Collection of values referenced by class members or annotations.
  - Includes primitive values, strings from literal expressions, and names of types and methods used in the class.
- **Field list:**
  - Contains a list of all declared fields, specifying their type, name, and modifiers.
- **Method list:**
  - Contains a list of all declared methods, including both abstract and non-abstract methods.
  - Non-abstract methods are described by byte-encoded instructions, representing the method's Java bytecode.

These and further details are encoded in a binary format into `.class` files.

2.7.2 Jasmin

Jasmin was developed in response to the absence of an official assembler format for the Java Virtual Machine (6). Generating a binary `.class` file manually is a complex task, similar to creating an executable file like `.exe` by hand. Although tools like `JAS`, a Java API for generating class files - used internally by Jasmin - simplify the process, they still require an understanding of the JVM's internal workings, such as the constant pool and attribute tables.

A Jasmin file typically consists of:

- **Class Declaration**: Declares the class with `.class` followed by the class name.
- **Fields Declaration**: Declares fields with `.field` keyword.
- **Method Declaration**: Each method starts with `.method`, specifying its name, access flags, and signature.
- **Instructions**: JVM bytecode instructions that interact with the stack and the local variable array.

2.7.2.1 Operand Stack

The operand stack in Jasmin is a temporary storage area used by instructions to hold values and perform calculations. Each method invocation has its own operand stack. Instructions such as `ldc`, `iload`, and `iadd` manipulate the stack by pushing, popping, or operating on values. The stack size can be set using the `.limit stack` directive. However, since determining an exact maximum size is difficult, it is generally more reliable to estimate an upper bound conservatively.

2.7.2.2 Local Variable Array

The local variable array is a fixed-size array where each method stores its parameters and local variables in dedicated *slots*. This array is indexed by position, starting at index 0. Calculations are achieved by loading values into the operand stack and manipulating them there. The array size can be set through the `.limit locals` directive and an upper bound should be estimated conservatively. Every `Variable_declarationContext` instance has a field `index` that holds the current position of the slot of the variable in its local array.

2.7.2.3 Labels

Labels in Jasmin mark specific points within a method's bytecode, serving as targets for jump instructions such as `goto`. They allow for control flow constructs, such as loops and *if-else* statements. A label is defined by typing a name followed by a colon (`:`) and is scoped to the method in which it is defined.

Here is an instructive example showcasing these prominent features:

Jasmin Example

```
.method public static sumOfN(I)I
    .limit stack 2             ; Maximum stack size needed
    .limit locals 2            ; Maximum array size needed
                               ; (input n + current sum)

    iconst_0                   ; Push 0 onto the stack
    istore_1                   ; Store 0 in local_variable[1] (sum)

    iload_0                    ; Load input n onto the stack
loop:
    ifle end                   ; If n <= 0, jump to end label

    iload_1                    ; Load current sum onto the stack
    iload_0                    ; Load n onto the stack
    iadd                       ; Add sum + n
    istore_1                   ; Store the new sum in local_variable[1]

    iinc 0 -1                  ; Decrement n
    goto loop                  ; Jump back to loop

end:
    iload_1                    ; Load the final sum from local_variable[1]
    ireturn                    ; Return the sum
.end method
```

2.7.3 Bytecode Generator

Essentially, Jasmin accepts an arbitrary input string of instructions that is then compiled to Java Bytecode. To construct the string, it again suffices to traverse the syntax tree with another visitor called the `BytecodeGenerator`. This time, the visitor methods do return strings so that the input string is built recursively using a `StringBuilder` instance for every node. For example, the bytecode generation for an `if`-statement looks like this:

BytecodeGenerator.java

```java
...
@Override
public String visitStatement(GrammarParser.StatementContext ctx) {
    StringBuilder sb = new StringBuilder();
    ...
    if (ctx.IF() != null) {
        // Generate two unique labels for branching
        String labelElse = generateLabel();
        String labelEnd = generateLabel();

        // Append the boolean expression string
        sb.append(visitExpression(ctx.expression()));

        sb.append("   ifeq ").append(labelElse).append("\n");

        // Append the bracket statements if clause evaluates to true
        sb.append(visitBracket_statement(ctx.bracket_statement(0)));
        sb.append("goto " + labelEnd + "\n");

        sb.append(labelElse + ":" + "\n");
        // Append the bracket statements if clause evaluates to false#
        // and 'else' is defined
        if (ctx.ELSE() != null) {
            sb.append(visitBracket_statement(ctx.bracket_statement(1)));
        }

        sb.append(labelEnd + ":" + "\n");

        return sb.toString();
    }
...
```

2.8 Optimisation

It is now possible to map the Mini Java code to Java bytecode directly. However, since the compiler produces executables that in a production setting will be run over and over again by the JVM, it is generally worthwile to consider optimising code already during its compilation. Since compilation is an annoyance to the developer and of little concern to the user, spending a few seconds extra to generate more space and runtime efficient bytecode is reasonable. It must be noted that the JVM does applies more optimisation techniques than compilers targeting the JVM, since the JIT enables more aggressive optimsation (and deoptimisation if the optimsation of the code turns out to be incorrect) by incorporating runtime information.

Nonetheless, some work can be lifted from the JVM's shoulders by implementing basic optimisation techniques. The optimisation intended for MiniJava deals with the immediate evaluation of constant expressions and replacement of variable expressions by their assigned literal value.

2.8.1 Constant Folding

A seemingly simple optimisation that serves as a stepping stone for further optimisation techniques. An expression that is made up of only `int` and `boolean` literals can be simply be evaluated during compilation.

Since expression evaluation must be done recursively, it integrates well with typechecking. Each expression either returns `null` if it cannot be folded into a constant or an `Integer` instance representing the folded value of the expression.

In addition to reducing the size of expressions, folding also allows for the elimination of one branch in `if` or `while` statements, simplifying conditional logic.

2.8.2 Constant Propagation

It is reasonable to question whether only literals should be considered foldable or if other literal value holders, such as variables, could also be folded if their values are tracked throughout the compilation process and replaced when expressions contain them. This approach, known as *constant propagation*, complements the initial optimization by allowing the compiler to fold any expression not involving a method invocation, as such values generally cannot be determined at compile time.

OptMyVisitor.java

```
...
@Override
public Object visitStatement(GrammarParser.StatementContext ctx) {
    String type;
    Integer fold;
    if (ctx.IF() != null || ctx.WHILE() != null) {
        fold = typeCheck(ctx.expression(), SimpleType.BOOL);
        ctx.expression().setFold(fold);
    }
...
```

To implement this, a field `fold` of type `Integer` is added to the expression node, which is set after folding. During bytecode generation, this field is checked and read before the bytecode for non-foldable expressions is explicitly generated.

OptTypeChecker.java

```
...
public static Integer checkBinaryExpression(
ExpressionContext ctx,
Object expectedType
)
{
...
// 'less than' expression
if (ctx.LT() != null) {
    Object actual = SimpleType.BOOL;
    if (!actual.equals(expectedType))
        ExceptionUtils.throwException(...);

    Integer arg1 = typeCheck(ctx.expression(0), SimpleType.INT);
    Integer arg2 = typeCheck(ctx.expression(0), SimpleType.INT);

    // Check if both sub expressions yield folded values
    if (arg1 != null && arg2 != null) {

        // Fold the current boolean expression
        Integer result = arg1.compareTo(arg2);
        return (result > 0) ? Integer.valueOf(0) : Integer.valueOf(1);
    } else
        return null;
}
...
```

3 Conclusion

A compiler has been developed for MiniJava, an educational subset of the Java programming language, showcasing all compiler stages and optimization.

Initially, the thought of generating the parser with ANTLR seemed promising. However, ANTLR provides limited flexibility for optimization, as any operations outside the parse tree traversal introduce substantial overhead. Defining the grammar in ANTLR's context-free grammar format limits customization: It barely allows modification of syntax tree nodes before generation. Consequently, adding fields and methods to nodes must be done manually or through additional scripts or build tools.

Still, ANTLR serves as a valuable learning tool for compiler construction. With the knowledge and experience now gained and more time, it seems preferable to implement a handwritten parser, as it would allow much greater flexibility in error reporting and optimization techniques.

On the other hand, Jasmin has proven to work delightfully, placing minimal restrictions on the programmer and not requiring deep expertise regarding JVM internals. This contrasts with alternatives like `ByteBuddy` or `ASM`, which require advanced JVM knowledge and are much harder to setup, but admittedly do offer a wider range of features.

A positive takeaway has been that the compiler's construction only required a few data structures of moderate complexity to achieve results, with limited dependencies between stages. Building a compiler draws on much of the knowledge covered in an undergraduate computer science curriculum, from software engineering and theoretical computer science to computer architecture, making it a highly recommendable academic project.

A MiniJava Grammar

```
class_declarations main_class EOF;

class_declarations: class_declaration*;

main_class:
    PUBLIC CLASS 'Main' '{' method_declaration* main_method '}';

main_method:
    PUBLIC STATIC VOID MAIN '(' ')' '{' statements '}';

class_declaration:
    CLASS IDENT '{' declarations '}';

declarations: declaration*;

declaration: variable_declaration | method_declaration;

method_declaration:
    method_type IDENT '(' ')' '{' statements RETURN expression ';' '}'
    | VOID IDENT '(' formals ')' '{' statements '}'
    | method_type IDENT '(' formals ')' '{' statements RETURN expression ';
    | VOID IDENT '(' ')' '{' statements '}';

formals: (type IDENT (',' type IDENT)*)?;

statement:
    local_variable_declaration
    | IF '(' expression ')' bracket_statement (ELSE bracket_statement)?
    | WHILE '(' expression ')' bracket_statement
    | PRINT '(' expression ')' ';'
    | lvalue ASSIGN expression ';'
    | method_invocation ';';

statements: statement+;

bracket_statement: '{' statements '}';

local_variable_declaration: variable_declaration;

variable_declaration: type IDENT ';';

lvalue: field_access | array_access | IDENT;
```

```
field_access: 'this' '.' IDENT | IDENT '.' IDENT;

array_access: IDENT '[' expression ']';

array_type: simple_type '[]';

method_invocation:
    IDENT '.' IDENT '(' method_arguments ')'
    | 'this' '.' IDENT '(' method_arguments ')'
    | IDENT '.' IDENT '(' ')'
    | 'this' '.' IDENT '(' ')';

method_arguments: expression (',' expression)*;

expression:
    IDENT
    | NUMBER
    | '-' expression
    | expression ('+' | '-' | '*' | '/' | '%') expression
    | '(' expression ')'
    | IDENT '.' LENGTH
    | array_access
    | 'new' simple_type '[' expression ']'
    | 'new' IDENT '(' ')'
    | expression ('&&' | '||' | '<' | '==' | '!=') expression
    | '!' (expression)
    | 'true'
    | 'false'
    | field_access
    | method_invocation;

type: simple_type | array_type;

simple_type: 'int' | 'boolean' | IDENT;

method_type: 'int' | 'boolean' | 'void' | IDENT;
```

References

[1] D. Jones. (2017) Evidence for 28 possible compilers in 1957. Accessed: 2024-10-29. [Online]. Available: https://shape-of-code.coding-guidelines.com/2017/05/21/evidence-for-28-possible-compilers-in-1957/

[2] A. Snyder, "A portable compiler for the language C," Master's thesis, MIT, 1975.

[3] ORACLE. (1999) 1.2 design Goals of the Java Programming Language. Accessed: 2024-10-29. [Online]. Available: https://www.oracle.com/java/technologies/introduction-to-java.html

[4] ANTLR. (2024) What is ANTLR? Accessed: 2024-10-29. [Online]. Available: https://www.antlr.org/

[5] B. Venners. (2021) Inside the Java Virtual Machine. Accessed: 2024-10-29. [Online]. Available: https://artima.com/insidejvm/ed2/index.html

[6] J. Meyer. (1996) Jasmin Instructions. Accessed: 2024-10-29. [Online]. Available: https://jasmin.sourceforge.net/instructions.html