JOHANNES KEPLER
UNIVERSITY LINZ

Author
Paul Lehner
K12145051

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Christoph Pichler, BSc

September 2024

Abstract Syntax Trees for MicroJava

Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

A typical process of a compiler involves translating source code into an intermediate representation, e.g. an Abstract Syntax Tree (AST). This intermediate representation can then be further processed. However, the conversion from source code to an AST as well as how the program is interpreted can be challenging to follow. This thesis addresses this issue by developing an AST interpreter and visualizing the corresponding program in its AST representation. During compilation, the program is translated into an AST. For program execution, this AST is traversed, with the interpreter embedded directly within the nodes of the AST. As the tree is traversed, it is also translated to a graph description string. By visualizing this AST alongside the source code, an interactive AST viewer is implemented, providing greater insight into how an AST interpreter operates. In this thesis, an AST interpreter and visualizer for MicroJava, a simplified version of Java, were implemented.

Contents

Chapter 1

Introduction

In today's modern world, technology is advancing at a tremendous pace, expanding the selection of programming languages. One language is MicroJava, a simplified version of Java used at JKU for educational purposes. Like Java, MicroJava is typically compiled to byte code [1]. However, in a different approach, MicroJava will be compiled using an abstract syntax tree (AST). The basis of the approach in this thesis is the book "Compilerbau" [2] by Hanspeter Mössenböck. In his book he presents the approach of building up an AST based on source code and traversing the tree to interpret the program directly without the usage of byte code. Such a process can be complex and difficult to follow, particularly for those new to the field.

This thesis aims to address that issue. To create a compiler skeleton for MicroJava, the compiler generator CoCo/R is used to automatically generate the scanner and parser. Along with a semantic specification, the parser inserts the appropriate nodes for the corresponding code segments to construct the AST. By traversing the entire tree, the AST generates a graph representation, which is then visualized using Graphviz and JavaFX.

The following chapters are structured as follows: Chapter 2 introduces the background, including existing concepts and frameworks. Chapter 3 provides an introduction to the MicroJava language. Chapter 4 focuses on the implementation, particularly how the different types of graph nodes were implemented. Chapter 5 demonstrates the entire implementation through a usage example. Finally, Chapter 6 discusses the conclusions of this thesis and potential future optimizations.

Chapter 2

Background

For the implementation of the AST compiler, the interpreter and the visualisation of the AST, some existing concepts and frameworks are used and will be presented in the following.

2.1 Abstract Syntax Tree

Abstract Syntax Trees (ASTs) serve as an intermediate representation of program code during the compilation process and are also commonly used for visualization [3, 4]. In general, ASTs simplify the process of analysing and transforming code by organizing it hierarchically according to its syntactic structure. Each node in the AST represents a syntactic element of the source code, such as variables, operators, or control structures (like loops and conditionals). The tree structure reflects the nesting and ordering of these elements in the code.

The AST can be created by defining a fixed order of priory. The lowest priority is the outermost node and the deeper it is nested, the higher is the priority. The expression $a = b+c$, for example, can be expressed as an AST as shown in Figure 2.1. The lowest priory has the = sign, which assigns the left child a with the result of the right child. The right child + is the parent of b and c which applies the operator on these child nodes. So, at first, addition must be executed before the assignment can be done, which preserves the order of execution.

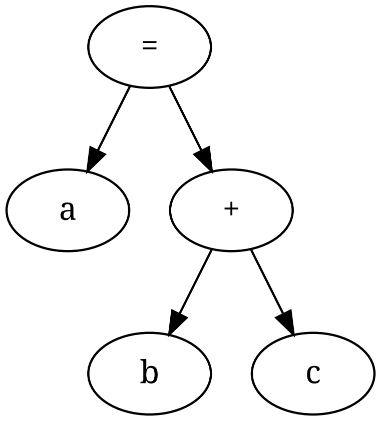


Figure 2.1: AST resulting from the statement $a = b + c$

2.2 CoCo/R

CoCo/R [5] (a **co**mpiler **co**mpiler that generates **r**ecursive descent parsers) is a compiler generator originally developed by Hanspeter Mössenböck. As shown in Figure 2.2, CoCo/R takes an EBNF-grammar as compiler description and automatically generates the foundational structure of the scanner and parser.

The resulting structure can verify the lexical and syntactical correctness of an input program. However, it is not yet capable of interpreting the code or checking for semantic errors, as these aspects heavily depend on the specific language specification. To achieve the desired functionality, CoCo/R must be attributed. This can be done by not only providing an EBNF-grammar as input, but an attributed grammar (ATG) [6]. These specifications are formalized in an `.atg` file. The `atg` file used in this thesis is referred in Section 4.5. A detailed explanation of CoCo/R is available in the user manual [7].

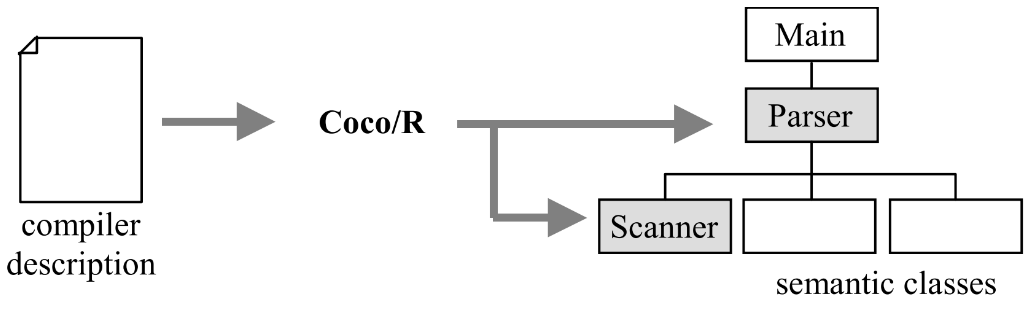


Figure 2.2: Description of CoCo/R [7]

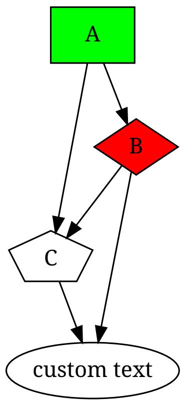
2.3 DOT Language and Graphviz

The DOT language is a plain-text graph description language [8]. It allows users to define graphs, nodes and edges, specifying their properties and relationships. DOT supports directed and undirected graphs. An example of a DOT specification of a directed graph is shown in Figure 2.3a: Various properties can be defined like background colour, shape of nodes, and custom label names for nodes. Edges are declared in lines 2 to 4, whereas nodes and their properties are written in lines 5 to 8.

To draw such a DOT graph, Graphviz can be used. Graphviz is an open-source graph visualization software tool that generates visual representations of graphs based on graphs specification in the DOT language format. It comes with several layout engines and allows to export the resulting graph in various output formats [9]. Applying Graphviz to the DOT specification example in Figure 2.3a results in the graph in Figure 2.3b.

```
1 digraph G {
2   A -> {B, C}
3   B -> {C, D}
4   C -> D
5   A[shape=rectangle, style=filled, fillcolor=green]
6   B[shape=diamond, style=filled, fillcolor=red]
7   C[shape=pentagon]
8   D[label="custom text"]
9 }
```

(a) Example graph in DOT language



(b) Resulting graph

Figure 2.3: DOT example and its resulting graph generated by Graphviz

2.4 JavaFX

JavaFX is an open-source client application platform for desktop built on Java and is the standard for building an interactive GUI interface in Java [10]. The concept of JavaFX is based on a scene graph. Every element of a JavaFX application is defined as a node. By appending a JavaFX element to another element, a graph hierarchy is built. One of many elements are for example buttons, text fields or image viewers. Each of those elements can handle various events like detecting key strokes or mouse clicks. On those events functions can be called to achieve interactive logic. Though it is possible to build the scene graph entirely by hand in source code, external visualization tools like SceneBuilder can be used for designing the application [11]. Figure 2.4 illustrates a basic GUI of a JavaFX application. The root node is a `VBox`, which arranges its child nodes vertically within the scene. In this example, a `Label`, a `Button` and a `TextField` are added to the `VBox`.

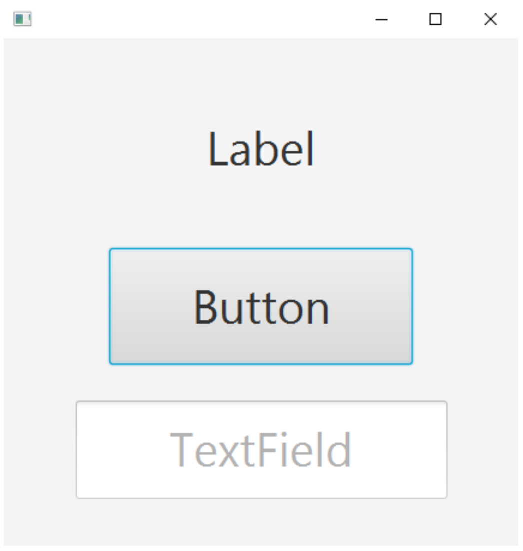


Figure 2.4: Basic JavaFX GUI with four elements

Chapter 3

MicroJava

MicroJava is a lightweight version of Java used to teach the fundamentals of compiler design. It shares many similarities with Java, including the use of byte code and a virtual machine for execution, but it simplifies several aspects to make it easier for beginners to grasp. It is mainly used for educational purposes at Johannes Kepler University. The MicroJava language is defined using an EBNF grammar [12] for its syntax and by the corresponding semantic specification [13]. The grammar is attached at the end of this thesis. In MicroJava it is possible to

- define global variables as well as local variables
- store values in two primitive types, `int` and `char`
- define custom `classes`
- define `arrays` of all types
- call methods with arbitrary parameters and primitive return types.
- do conditional jumps with `if-else` and `while` statements
- use in-build methods like `print` and `read` for user interaction

As mentioned before, the MicroJava compiler generates byte code. This code is a command set of elementary operations, like memory manipulation or arithmetic operations. MicroJava is specified with a command set of 57 byte codes [13].

3.1 MicroJava VM

The MicroJava VM is a virtual machine (VM) which executes the byte code of a MicroJava program. Like the Java VM, it is a stack machine, i.e. temporary values are loaded onto an expression stack. The MicroJava VM is primarily based on the code skeleton from the exercise class of the compiler construction lecture. Its main task is to manage the memory for MicroJava. There are five different memory regions as shown in Figure 3.1.

- *Code array*
  The code array provides the program, described in MicroJava byte code format. The virtual machine interprets each command and executes the corresponding task.

- *Data stack*
  This stack stores global variables. Since the number of global variables does not change during execution, the stack size is fixed.

- *Heap*
  The heap stores reference objects. It has space for 10,000 values, with the zero address reserved for the null reference.

- *Proc or method stack*
  The method stack stores the local variables of the currently executed method. When a method is called, the VM allocates space for all variables of the callee and frees it upon return. In MicroJava, there is space for 4,000 variables.

- *Expression stack*
  The expression stack is used for temporarily storing intermediate results. For example, when performing addition, the two numbers are loaded onto the stack and then combined to a single result. As the stack has space for only 30 values, no value should remain on the stack after computation (a critical detail for Section 4.3.5).

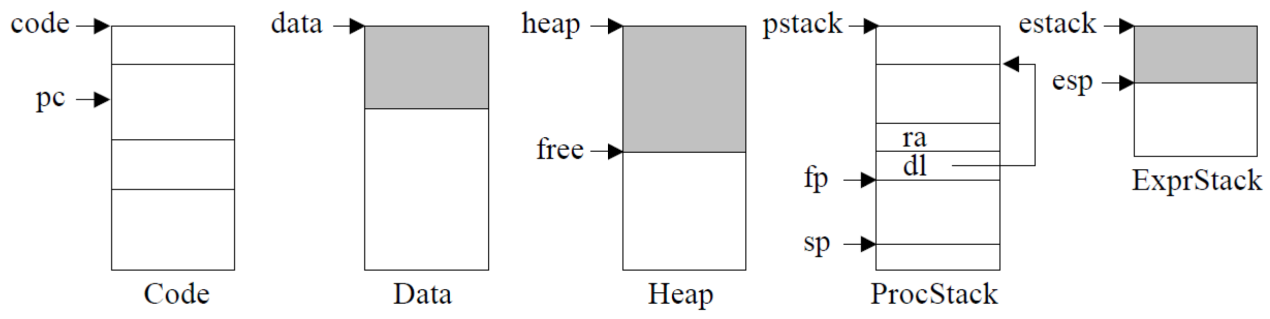


Figure 3.1: MicroJava memory layout [13]

Chapter 4

Implementation

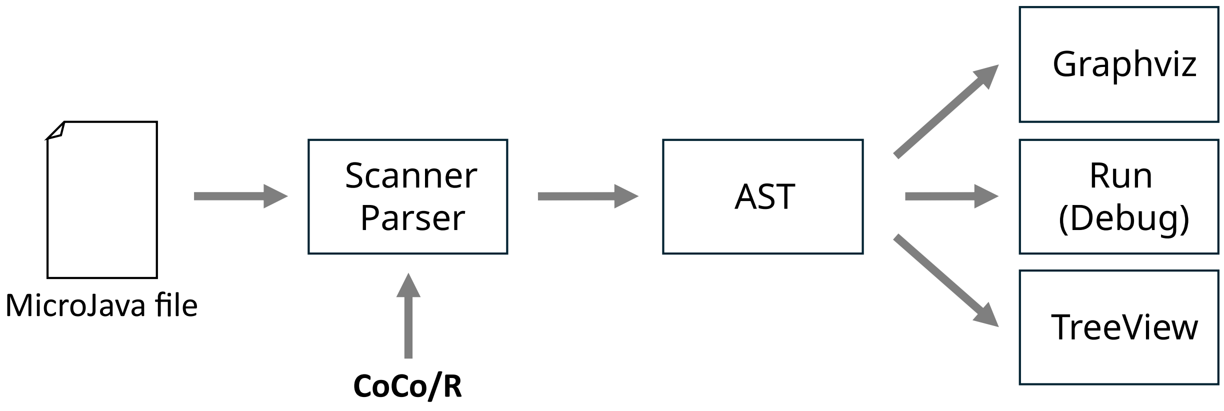


Figure 4.1: Workflow of the implementation

The primary objective of this thesis is to construct the syntax tree for the programming language MicroJava. The structure of the implementation is shown in Figure 4.1. As input, the program receives a MicroJava file. The scanner and parser check if the file is a valid MicroJava program. As mentioned in Section 2.2, for generating the scanner and parser, CoCo/R is used. With the attributed grammar, the parser is capable of setting the right nodes at the needed position to build up the AST. Based on the AST, a DOT representation of the AST can be generated and with the Graphviz engine converted to an image. Additionally, the AST will be translated to a JavaFX `TreeView` element. An interpreter can either execute the AST all at once, or by entering the debug mode and executing the program step by step. During this, the current program line as well as the current node gets highlighted in both AST visualizations.

4.1 Node Hierarchy

The node hierarchy defines the inheritance order of all nodes used in the AST. In an AST, every expression and statement is represented as a node object as shown in Figure 4.2. The base class for all these nodes is shown in Listing 4.1. The fundamental operations of every node are described in the following:

- The *execute* method receives an interpreter instance as parameter. The interpreter manages the memory and provides access to the program's heap, stack, global variables, and expression stack. For this, the MicroJava VM mentioned in Section 3.1 is used. However the *code array* is not used, since the code instructions are given by the context of the graph execution. Depending on the current node, the implementation of the *execute* method defines the correct instruction. For control flow purposes, every *execute* method must be capable of throwing a `ControlFlowException`, which will be discussed in Section 4.4.

- The *toDOTString* method traverses the tree and generates a string representation of the AST in the graph description language DOT. The string gets inserted into the `StringBuilder` object *sb*. As shown in Listing 2.3a, the edges between nodes are defined by `parentName -> childName`. Given the parameter `parentName`, each node can introduce its child relation to the string description. Because the name of each node must be unique, the names are defined in the constructor as consecutive numbers, given by the static `count` variable of the `Node` class. On every node instance `count` gets incremented.

- The *toTreeView* method builds up a `TreeItem` tree, for the JavaFX `TreeView` element. Each node calls its child nodes to retrieve their `TreeItem`. After adding the child to the tree a new `TreeItem` is returned.

- Every node must implement the *getName* method to define its name. The names of the nodes in the visualization are given by this method.

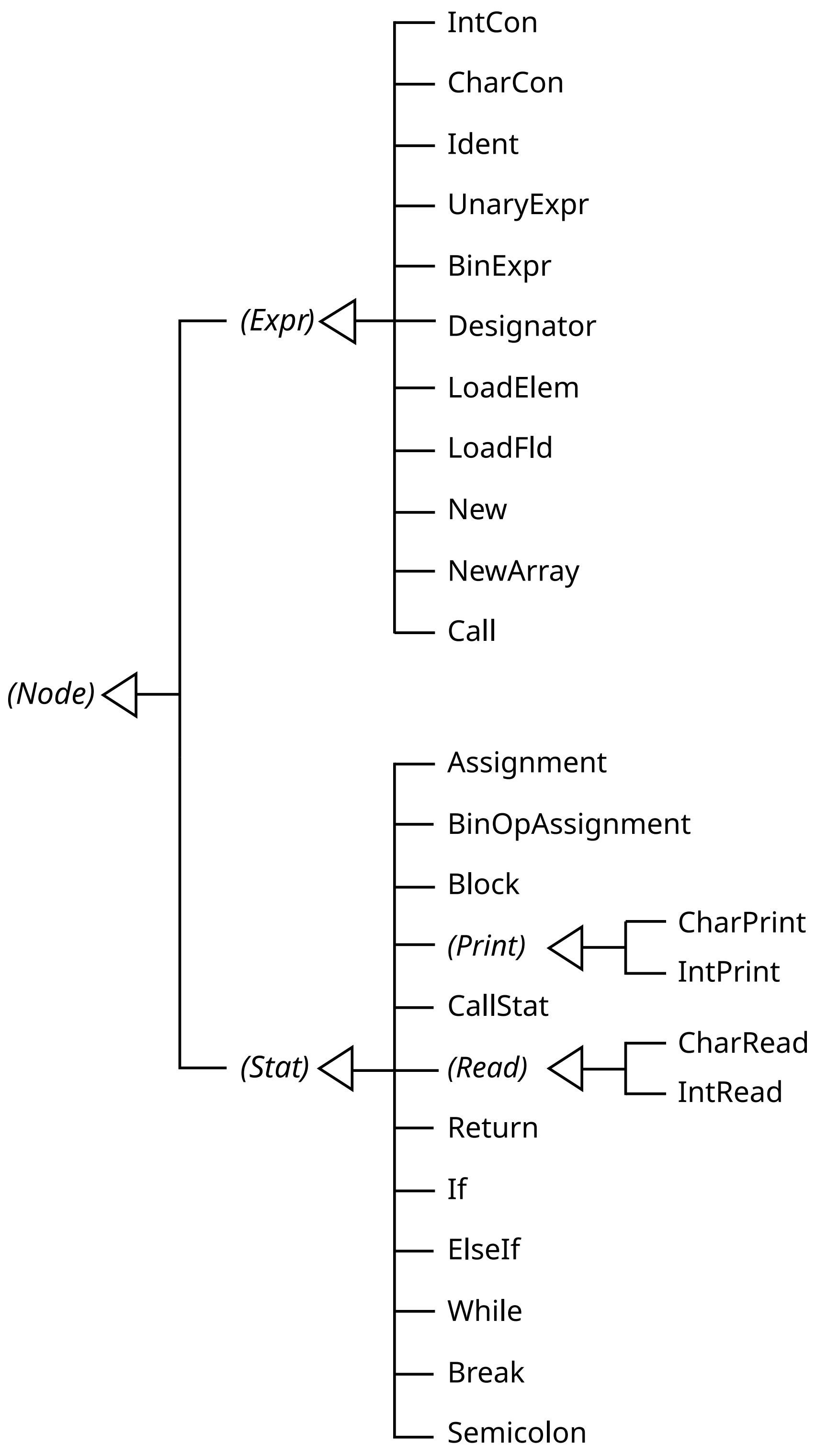


Figure 4.2: Class Hierarchy of the AST nodes

```java
public abstract class Node {
  ...
  public void execute(Interpreter interpreter) throws ControlFlowException
    { ... }
  public void toDOTString(StringBuilder sb, String parentName) { ... }
  public TreeItem<Node> toTreeView() { ... }
  public abstract String getName();
  ...
}
```

Listing 4.1: Abstract base class for all nodes

4.2 Expression Nodes

Expression nodes represent different kind of expressions. When computing expressions, one critical aspect is the data type. The class `Expr`, as shown in Listing 4.2, has the following fields:

- *type*
  In MicroJava, it is not possible to add an `int` and a `char` together, and implicit type casting is not available in MicroJava. Therefore, for the compiler to verify the correct types of an expression, information about the object type must be provided. To facilitate this, every expression node has a `Struct` field that contains all the necessary information for type checking.

- *offset*
  The *offset* field is required to calculate the absolute memory address.

- *kind*
  Since not every variable address can be accessed using the same procedure, like an array, the *kind* of the variable must be stored.

4.2.1 Leaf Node

Leaf nodes are nodes that do not have child nodes themselves. They represent atomic expressions like constants and variables.

```java
public abstract class Expr implements Node {
  public enum Kind {
    Con, Local, Static, Fld, Elem, Meth, None
  }
  public Struct type;
  public int offset;
  public Kind kind;
  ...
}
```

Listing 4.2: Expression node

- *Primitive Con*

  During the parsing process, when a constant token of type `int` or `char` is encountered in the code, the parser inserts an `IntCon` node or a `CharCon` node, respectively. Their primary purpose is to store the corresponding value as well as the `string` representation, which will be used for tree visualization. The *execute* method simply loads the value onto the expression stack, where it will be further processed by its parent nodes.

- *Ident*

  `Ident` nodes represent variable accesses, methods and the start node of the program. An `Ident` node wraps an `Obj` node, which contains all the necessary information about the variable. The program node executes the main node, as shown in Listing 4.3 at line 7. All methods execute their block node, as shown in line 10. A constant node pushes its value onto the expression stack in line 13. There are four kinds of variables: *locals*, *globals*, *fields*, and *arrays*. In all cases, the *execute* method of the node loads the value stored in the variable onto the expression stack, which depends on the type of the variable. For instance, if the variable is an array, the node expects that both the address and the index are already on the stack, so it can retrieve them in lines 24 to 25 and push the correct element back onto the stack in line 26.

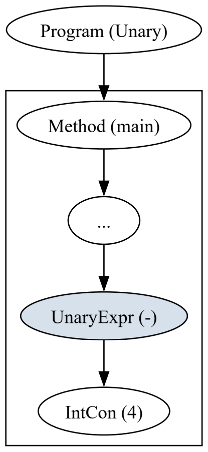
```java
public class Ident extends Expr {
  ...
  @Override
  public void execute(Interpreter interpreter) throws ControlFlowException {
    super.execute(interpreter);
    switch (obj.kind) {
      case Prog:  main.execute(interpreter);
                  break;
      case Meth:  ...
                  block.execute(interpreter);
                  ...
                  break;
      case Con:   interpreter.push(obj.val);
                  break;
      case Var:   int adr, idx;
        switch (kind) {
          case Local:  interpreter.push(interpreter.getLocal(var.adr));
                       break;
          case Static: interpreter.push(interpreter.getData(var.adr));
                       break;
          case Fld:    adr = interpreter.pop();
                       interpreter.push(interpreter.getHeap(adr + var.adr));
                       break;
          case Elem:   adr = interpreter.pop();
                       idx = interpreter.pop();
                       interpreter.push(interpreter.getHeap(adr+1+idx));
                       break;
        ...
        }
    ...
    }
  }
}
```

Listing 4.3: Execute method of `Ident` node

4.2.2 UnaryExpr Node

```
program Unary {
  void main() int a; {
    a = - 4;
  }
}
```

(a) Example MicroJava code



(b) AST representation

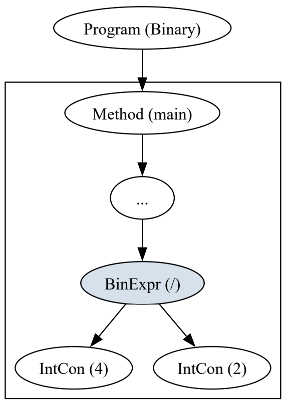
Figure 4.3: Unary example

The task of the `UnaryExpr` node is to apply an unary operation to an expression. In MicroJava, the only unary expression is negation with a minus sign, but in theory, additional operations such as logical not or bitwise negation could be supported the same way. In Figure 4.3, an example program is shown.

4.2.3 BinExpr Node

```
program Binary {
  void main() int a; {
    a = 4 / 2;
  }
}
```

(a) Example MicroJava code



(b) AST representation

Figure 4.4: Binary example

The `BinExpr` node is one of the most frequently used node in MicroJava and represents a binary expression. Additionally, it handles logical evaluations when the operator represents a condition operation. The node is assigned by the parser with two expressions and an operator. When the *execute* method is called (Listing 4.4), the `BinExpr` node first executes its left child, retrieving one value from the expression stack as seen in lines 5 to 6. There are two types of expressions:

- *Numerical expression*
  The computation is straightforward: the appropriate operation is applied, and the result is pushed onto the expression stack for further execution as for example for addition operation in line 21. In case of division or modulo operations, it is essential to check that the divisor is not zero. Otherwise, a "division by zero" exception is raised, like in line 25. MicroJava also includes a built-in power operation for positive integer exponents, which is computed using a loop.

- *Condition*
  When the operator is a logical expression, such as `and`, the results on the expression stack must be interpreted as boolean values. The convention used in this thesis follows the classical interpretation: 1 represents `true`, and 0 represents `false`. During evaluation, short circuit evaluation is applied. It is important to be aware of this technique, as it may prevent certain evaluations from being executed, even if they were intended to be by the programmer.

For example, the expression $a = 4/2$ from Listing 4.4a gets translated to the AST from Figure 4.4b. The `BinExpr` evaluates the left child and the right child by calling their *execute* methods. This is where the advantage of the AST becomes evident: regardless of the complexity of the left child's expression, the `BinExpr` node does not need to be aware of it. It simply receives the final value from its child, in this example the value 4. The same process applies to the right expression. The subsequent computation is then performed based on the assigned operator, in this example the division operator.

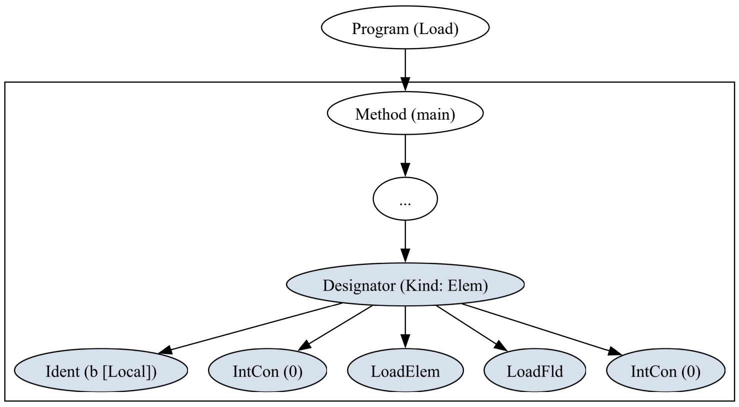
```java
public class BinExpr extends Expr {
  ...
  @Override
  public void execute(Interpreter interpreter) throws ControlFlowException {
    left.execute(interpreter);
    int lval = interpreter.pop();
    boolean blval = (lval == 1);
    if (op == Operator.OR && blval) {
      interpreter.push(1);
      return;
    }
    else if (op == Operator.AND && !blval) {
      interpreter.push(0);
      return;
    }
    right.execute(interpreter);
    int rval = interpreter.pop();
    boolean brval = (rval == 1);

    switch (op) {
      case ADD: interpreter.push(lval + rval);
                break;
      ...
      case DIV: if (rval == 0) {
                  throw new IllegalStateException("division by zero");
                }
                interpreter.push(lval / rval);
                break;
      ...
      case EQL: interpreter.push((lval == rval)? 1:0);
                break;
      ...
      default: throw new IllegalStateException("Operator not implemented");
    }
  }
  ...
}
```

Listing 4.4: Execute method of `BinExpr` node

4.2.4 Load Nodes

```
program Load
  class B { int[] a; }
{
  void main() B[] b; {
    b = new B[1];
    b[0] = new B;
    b[0].a = new int[1];
    b[0].a[0] = 4;
  }
}
```

(a) Example MicroJava code



(b) AST representation

Figure 4.5: Load example

Load nodes load values of reference-typed variables onto the expression stack. Reference types are accessed via their addresses on the heap. When a `class` node or an `array` node is executed, it places its address on the expression stack to grant access to the data. However, if a class instance has a field that needs to be accessed, the address is required only to retrieve the value stored in the field. There are three different types of `Load` nodes:

- `Designator` node:
  This node represents the entire object chain of a reference variable, for example `a.b[2].c.d[5]`. It contains a list of expression nodes arranged from left to right. In the *execute* method, all child nodes are looped through and executed sequentially.

- `LoadElem` node:
  To access an array element, two pieces of information are always required: the address of the array on the heap and the index of the element. The task of this node is to combine these two pieces of information and load the corresponding value from the array.

- `LoadFld` node:
  For accessing a field from an object, the `LoadFld` node is necessary. The `LoadFld` node retrieves the address from the expression stack and, with the help of the symbol table, returns the correct value from the object's field.

For example, in line 8 of Listing 4.5a, the zero element of the field in the zero element of the `class B` array is assigned the value 4. First, the local variable `b`, which is an array of `class B`, is accessed at position 0. Note that these two operations are represented by the `Ident` node and the `IntCon` node. The issue here is that the expression stack now contains the address of `b` and the value 0 as the index. However, for the next step, the address of the field array `a` is required. Therefore, both the address and the array index must be combined to access the instance of `class B`, which will yield another address. To achieve this, a load node is inserted at this position during the build up process of the AST. Since the outer construct is an array, a `LoadElem` node is inserted. During runtime, this places the address of an instance of `B` on the stack. However, the value required is the address of `a`, not the address of the instance itself. Since `a` is a field of `B`, a `LoadFld` node is used here. Finally, the index for `a` is also placed on the stack.

In this example, the reference type is assigned a value, so no further loading is necessary. The remaining work is handled by the `Assignment` node. However, if the same expression were on the right-hand side of an assignment, the entire expression would need to be loaded again with a `LoadElem` node, as the assignment would require the value to be stored in the variable, not its address and index.

4.2.5 Allocation Nodes

Allocation nodes allocate the memory of classes and arrays and push the assigned address onto the expression stack. The keyword `new`, like in Figure 4.5, indicates that space on the heap should be allocated for the corresponding data type. There are two types of `Allocation` nodes:

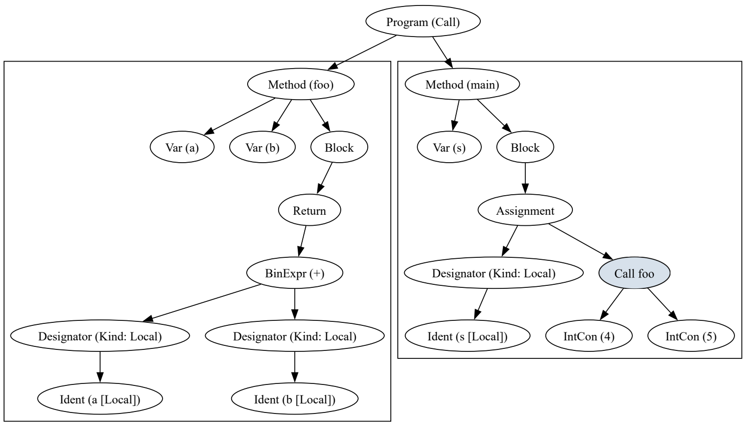
- `New` node
  This node is used for `class` objects. It allocates memory according to the number of fields in the data type and pushes the corresponding address onto the expression stack.

- `NewArray` node
  When a `NewArray` node is executed, the *size* of the array is evaluated. Subsequently, *size* + 1 units of space are allocated, with the *size* being stored at position 0. The address immediately following the *size* is then pushed onto the expression stack.

4.2.6 Call Node

```
program Call {
  int foo(int a, int b) {
    return a + b;
  }
  void main() int s; {
    s = foo(4, 5);
  }
}
```

(a) Example MicroJava code



(b) AST representation

Figure 4.6: Call example

To call a function, a `Call` node is inserted into the AST. Its tasks are to load the function's parameters onto the expression stack, set up the method stack, execute the function, and free the method stack upon returning. In Figure 4.6 the function *foo* is called. Consequently, the AST has the `Call` node *foo* with two constant nodes, 4 and 5, as the right child of the `Assignment` node. After loading these values onto the stack, the left box in Figure 4.6b is executed. The variables `a` and `b` are assigned from the expression stack. The function executes the `Return` node. In this example, the return value is the result of a `BinExpr` node. When the `Return` node executes the expression, the result will be on the expression stack for further processing in the *main* method.

By executing the `Call` node, the parameters are processed and subsequently loaded onto the expression stack, as shown in Listing 4.5 in line 7. After this, the methods are classified into four types, which are organized into three distinct cases:

- *ord* method and *chr* method
  These are built-in functions in MicroJava. The *ord* method converts a `char` type to an `int` type, while the *chr* method converts an `int` type back to a `char`. If the callee is either of these methods, the *execute* method does not perform any additional operations. The parameter itself serves as return value, only interpreted as a different data type.

```java
public class Call extends Expr {
  ...
  @Override
  public void execute(Interpreter interpreter) throws ControlFlowException {
    ...
    for (Expr para : paras) {
      para.execute(interpreter);
    }
    if (methodToCall.getObj() == Tab.ordObj ||
        methodToCall.getObj() == Tab.chrObj) {
      //nothing -> parameter is return value
    }
    else if (methodToCall.getObj() == Tab.lenObj) {
      int adr = interpreter.pop();
      if (adr == 0) {
        throw new IllegalStateException("null reference used");
      }
      interpreter.push(interpreter.getHeap(adr - 1));
    }
    else {
      int psize = methodToCall.getObj().nPars;
      int lsize = methodToCall.getObj().locals.size();
      interpreter.allocMethodStack(psize, lsize);
      try {
        methodToCall.execute(interpreter);
      } catch (ReturnException exp) {
        //return case
        interpreter.freeMethodStack();
        return;
      }
      if (methodToCall.type != Tab.noType) {
        throw new TrapException("Method " + methodToCall.getObj().name + "
    has no return statement");
      }
      else {
        interpreter.freeMethodStack();
      }
    }
  }
  ...
}
```

Listing 4.5: Execute method of `Call` node

- *len* method
  In line 13, the *execute* method checks if the `Call` node corresponds to the *len* method, which returns the length of an array. If this is the case, the array address must already lay on the expression stack, which is stored in line 14. As noted in Section 4.2.5, the array size is stored at the address immediately preceding the array reference. The size is retrieved and then pushed onto the expression stack in line 18.

- *custom* method
  For any other scenario, a custom function is called. In lines 21 to 23, the method stack is populated with the parameters and local variables, after which the actual method is invoked. When a `ReturnException` is caught by the try-catch block (as discussed in Section 4.4), the return value has already been placed onto the expression stack, and the method stack is subsequently freed. If no exception is raised but the method has a return type other than `void`, a `TrapException` is thrown in line 32. If the method has no return value, meaning the caller executed the node as a statement (see Section 4.3.5), the method stack is freed in line 35.

4.3 Statement Nodes

Unlike `Expr` nodes, which evaluate and return values, `Stat` nodes represent program statements like assigning values to variables and altering the program's control flow, such as through if conditions or loops by manipulating the state of the virtual machine.

4.3.1 Assignment Node

One of the most frequently used features in any programming language is the ability to assign specific values to variables. The tasks of the `Assignment` node are to set-up the left side, evaluate the right side and assigning the expression to the left side. In the *execute* method, the expression on the ride hand side is executed. Next, the result gets removed from the expression stack. In the next step, the left hand side of the assignment must be prepared. In case of *local* or *static* variables all needed information is given by the relative *offset* variable, so no preparation is needed. For *arrays* and *fields*, the left hand side must be resolved. By calling

```java
public class Interpreter {
  ...
  public void assign(Designator var, int val) throws ControlFlowException {
    int adr, idx;
    switch (var.kind) {
      case Local:  interpreter.setLocal(var.offset, val);
                   break;
      case Static: interpreter.setData(var.offset, val);
                   break;
      case Fld:    adr = interpreter.pop();
                   if (adr == 0) {
                     throw new IllegalStateException("null reference used");
                   }
                   interpreter.setHeap(adr + var.offset, val);
                   break;
      case Elem:   idx = interpreter.pop();
                   adr = interpreter.pop();
                   if (adr == 0) {
                     throw new IllegalStateException("null reference used");
                   }
                   int len = interpreter.getHeap(adr - 1);
                   if (idx < 0 || idx >= len) {
                    throw new IllegalStateException("index out of bounds");
                   }
                   interpreter.setHeap(adr + idx, val);
    }
    ...
  }
}
```

Listing 4.6: Assign method

the *execute* method of a variable, the address and index are pushed on the expression stack. At the end the *assign* method is executed to store the value.

Listing 4.6 shows the *assign* method. There are 4 different kinds of variables in MicroJava, where each of them requires its own way of performing assignments:

- *Local* variables
  If the variable `var` is stored on the method stack, the correct memory address can be determined by adding the offset of the variable to the frame pointer.

- *Static variables*
  Global variables get accessed in a similar way. Instead of the method stack, the data stack is used.

- *Field variables*
  By assigning a value to a field, the *assign* method expects the address of the variable already resolved and pushed on the expression stack. The address is received in line 10 and a check in line 11 against null references is preformed. Finally, the interpreter receives all the necessary information to perform the assignment, i.e. which value to write into which address of the heap.

- *Element variables*
  The memory access to array elements is comparable to field variables. Note that in case of an array two values are now expected, its address and the index of the element. After checking against null references and index out of bounds exceptions, the value gets assigned at the correct location on the heap.

4.3.2 BinOpAssignment Node

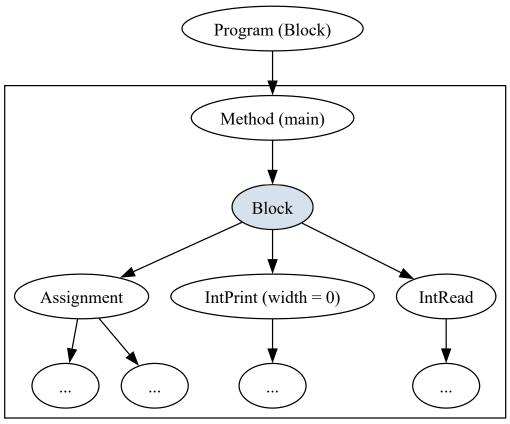
The `BinOpAssignment` node represents composition assignments like `+=` or `++`. On those statements, the variable occurs on the left hand side as well as on the right hand side of the assignment. For example, the statement `a[idx()]+=4` might seem equivalent to `a[idx()] = a[idx()] + 4`, but this is not the case. In a naive implementation, one would first evaluate the expression by adding 4 to the loaded `a[idx()]` variable. The same variable is executed again to resolve it for the *assign* method shown in Listing 4.6. So, `a[idx()]` got evaluated twice. This can lead to a dangerous side effect. If the *idx* method has side effects (i.e. if its execution changes global variables), the assignment violates the execution semantics. This can be seen directly if *idx()* returns different values for different calls (e.g. 0 for the first call and 1 for the second call). Since the expression is computed with for example `a[0]+4`, the left hand side of the assignment leads to `a[1]`, which results to the wrong statement `a[1]=a[0]+4`.

To overcome this issue, the `BinOpAssignment` only executes the left hand side of the composition once. For example, in case of an array, after executing the left hand side, the address and index are on the stack. Those values are used to load the corresponding value of the array element and performing the computation of the expression, based on the given operator. After that, both values are pushed back on the stack, leading to the same variable used for computation in the *assign* method instead of calling the left hand side of the composition again.

4.3.3 Block Node

```
program Block {
  void main() int a; int b; {
    a = 5;
    print(a);
    read(b);
  }
}
```

(a) Example MicroJava code



(b) AST representation

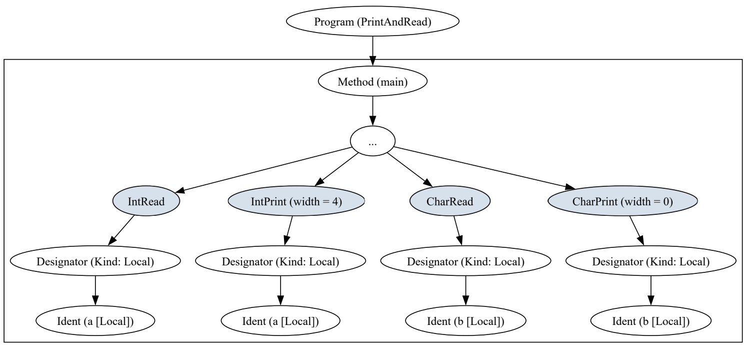
Figure 4.7: Block example

The `Block` node represents the curly braces for example of a method. The node contains all `Statement` nodes within the block, as shown in Figure 4.7, and executes them from top to bottom (or in case of an AST representation from left to right).

4.3.4 Print and Read Node

```
program PrintAndRead {
  void main() int a; char b;{
    read(a);
    print(a, 4);
    read(b);
    print(b);
  }
}
```

(a) Example MicroJava code



(b) AST representation

Figure 4.8: Print and Read example

In MicroJava, it is possible to print values using the built-in *print* method, which is implemented as two kinds of `Print` nodes, as shown in Figure 4.8. The two print nodes are `IntPrint`, which is used for printing integer values and `CharPrint` for printing characters. Optionally to the printing values, a second integer argument can be provided to specify the number of whitespace characters to include in the output. Similarly, there are two kind of nodes for reading user input from the console. The `IntRead` node reads an integer value, the `CharRead` reads a character.

4.3.5 CallStat Node

A `CallStat` node represents a function, called as a statement. If the called function returns a value, this value is ignored.

The *execute* method of the `CallStat` node is shown in Listing 4.7. Since the core functionality is identical to the `Call` node, the `CallStat` node simply wraps an expression `Call` node and executes it, as shown in line 5. If the method returns a value, it must be removed from the expression stack, as shown in line 7. Because the difference is purely logical, the AST representation of a statement call prints the `Call` node similarly to Figure 4.6b.

```
 1 public class CallStat extends Stat {
 2   ...
 3   @Override
 4   public void execute(Interpreter interpreter) throws ControlFlowException {
 5     methodToCall.execute(interpreter);
 6     if (methodToCall.type != Tab.noType) {
 7       interpreter.pop();
 8     }
 9   }
10   ...
11 }
```

Listing 4.7: Execute method of `CallStat` node

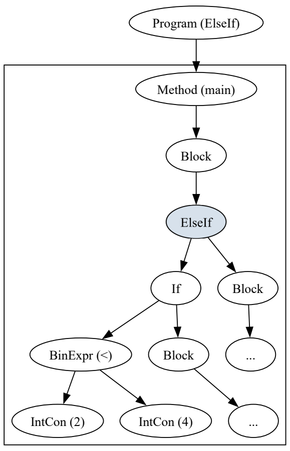
4.3.6 Return Node

The `Return` node is used to leave the called method and to optionally push the return value onto the expression stack by executing the given expression node. To jump back to the caller, a `ReturnException` is used, which will be explained in Section 4.4 in more detail.

4.3.7 ElseIf and If Node

```
 1 program ElseIf {
 2   void main() {
 3     if (2 < 4) {
 4       //do something
 5     }
 6     else {
 7       //do something else
 8     }
 9   }
10 }
```

(a) Example MicroJava code



(b) AST representation

Figure 4.9: ElseIf example

The `ElseIf` node is used for conditional checks. The node contains both an `If` node and a `Stat` node, where a `Stat` node represents the `else`-statement. First, the condition of the `If` node is evaluated. The result is interpreted as a truth value. Based on the result, either the statement block of the `If` node, as shown in Listing 4.9a in line 4, or the `Stat` node, as shown in 4.9a in line 7, gets executed. If the `If` condition does not have an associated `else` statement, only the `If` node is inserted into the AST (left child of the `ElseIf` node in Figure 4.9b).

4.3.8 While Node

The `While` node evaluates the loop condition and executes its `Statement` node repeatedly until the condition is no longer satisfied.

4.3.9 Break Node

The `Break` node represents a break statement within a loop. When the `Break` node is executed within a while loop, a `BreakException` is thrown. In the case of nested loops, this exception is caught by the innermost `While` node, causing it to terminate, which will be explained in more detail in Section 4.4.

4.3.10 Semicolon Node

By the definition of MicroJava, a semicolon by itself is considered a valid statement. For consistency, a corresponding node is inserted into the AST. However, since a semicolon has no operational effect, the *execute* method for this node is left empty.

4.4 Control Flow Exceptions

`ControlFlowExceptions` are used to deal with control flow statements. In cases of normal termination, execution returns to the caller due to the natural behaviour of recursive calls. However, if the flow is interrupted by, for example, a return statement, the *execute* method must jump back and continue executing in the previous execution branch. Therefore it is necessary to deviate from the standard tree traversal. One approach is to set a flag and check before each node whether a return statement has been raised. If so, the method should not execute further. In this thesis, the properties of exceptions are utilized. By throwing an exception at the return site, future statements are ignored until the node representing the call site catches it and continues execution from that point on. This technique is also used in Truffle, a framework for implementing managed language interpreter in Java [14].

There are three types of exceptions, derived from the `ControlFlowException` class, as shown in Figure 4.10. While the purpose of break and return exceptions, used in while loops and methods, is to control the program flow, the trap exception is raised when a function fails to execute a return statement. In such cases, the trap exception is triggered, terminating the execution. For example when a `Return` node is reached, the *execute* method throws a `ReturnException`, as shown in Listing 4.11a. The thrown exception is caught in Listing 4.11b by the `Call` node, which executed the method. The `BreakException` follows the same principle for break statements and while loops.

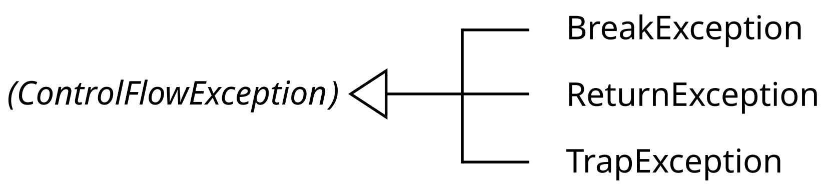


Figure 4.10: `ControlFlowException` Hierarchy

```java
public class Return extends Stat {
  ...
  @Override
  public void execute(Interpreter interpreter) throws ControlFlowException {
    ...
    throw new ReturnException();
  }
  ...
}
```

(a) Execute method of `Return` node

```java
public class Call extends Expr {
  ...
  @Override
  public void execute(Interpreter interpreter) throws ControlFlowException {
    ...
    try {
      methodToCall.execute(interpreter);
    }
    catch (ReturnException ex) {
      ...
    }
  ...
}
```

(b) Catch block of Call node

Figure 4.11: Throw and Catch example of a `ControlFlowException`

4.5 Attributed Grammar

An attributed grammar (ATG) is used to define the context for a grammar. The language of MicroJava is defined by a context free EBNF grammar, which specifies its syntactic structure. However, semantic context is also required, along with a specification of where each node should be inserted into the AST. Therefore, the purpose of the ATG is to perform semantic checks and facilitate the construction of the AST. This is achieved by embedding Java statements within the EBNF grammar, indicated by parentheses and dots, as shown in Listing 4.8. Based on this definition, CoCo/R automatically generates a parser that constructs a valid AST, which can then be executed.

```
 1 Term<out Expr expr>       (.Operator op; Expr expr2;.)
 2 = Factor<out expr>
 3 {
 4     Mulop<out op>
 5     Factor<out expr2>     (.if (expr.type != Tab.intType ||
 6                                 expr2.type != Tab.intType) {
 7                               SemErr("operand(s) must be of type int");
 8                             }
 9                             expr = new BinExpr(load(expr), op, load(expr2));
10                           .)
11     |
12     "**"
13     number                (.expr = load(expr);
14                             if (expr.type!=Tab.intType) {
15                               SemErr("operand(s) must be of type int");
16                             }
17                             expr = new BinExpr(expr, Operator.EXP, new IntCon(
    t.val));
18                           .)
19 }
20 .
```

Listing 4.8: Attributed grammar for `Term`

For example, Listing 4.8 shows a portion of the grammar for the non-terminal symbol `Term`. A `Term` returns an `Expr` node to the caller, indicated by the `out` keyword and the type `Expr` in line 1. A term is defined as a factor, followed by zero or more factors or exponential statements connected via multiplication operations. In line 2, the `Factor` production reads a factor and returns it as an `Expr` node. In case the next token is a `Mulop`, a second `Factor` is read in line 5. In case of a `**` token, a number is read in line 13. In both cases, the `Expr` node is replaced by a `BinExpr` node, in line 9 and 17. Note that in line 9 and 13, the expressions are wrapped around the helper method *load*, which inserts the correct load node as discussed in Section 4.2.4.

Additionally, during the AST construction process, the parser generated from the ATG checks in lines 5 to 8 that the two expressions are of type `int`. Otherwise a semantic error is raised. Similarly, lines 14 to 16 ensure the exponent is also an integer. The whole process repeats, until the next token is neither a `Mulop` or a `**` operation. In that case, the `Term` production is finished and returns the `Expr` node as a valid `Term`. This principle is applied throughout the MicroJava grammar specification, where the parser constructs the AST until the root node is returned.

4.6 JavaFX GUI

The GUI allows to load the source code, compiling it, visualize the corresponding AST and observe variables during runtime. The interface is shown in Figure 4.12.

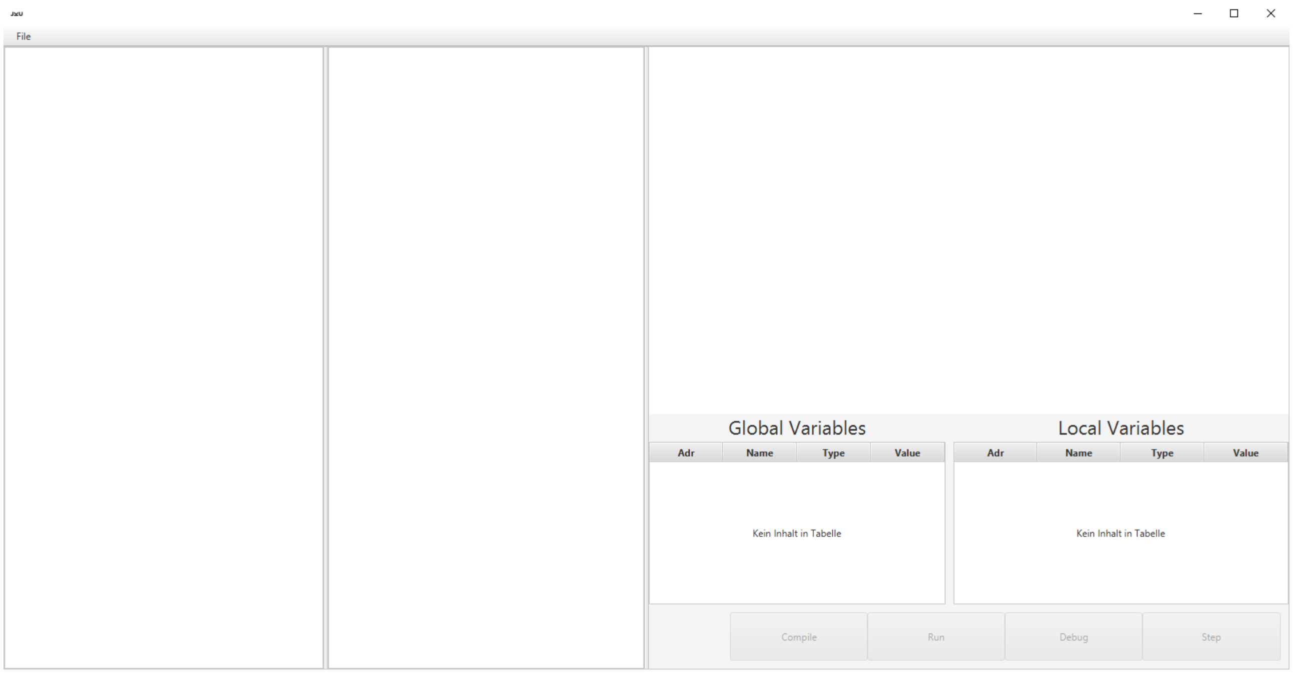


Figure 4.12: Interface of the AST interpreter implementation

The leftmost section contains a `ListView` element, which displays the source code line by line, along with corresponding line numbers. The second section features a `TreeView` element, which displays the nodes of the AST in a hierarchical structure. In the upper-right corner is a `WebView` element. By providing an HTML string, the `WebView` displays the `.svg` file generated by the Graphviz engine. Below the `WebView` element are two `TableView` elements that display the current state of both global and local variables by observing the interpreter instance during runtime. At the bottom, four buttons are available. The `Compile` button compiles the program. The `Run` button executes the program. The `Debug` button enters single step execution mode, where the program halts at each node. By pressing the `Step` button, the program executes one node at a time. In `Debug` mode, the AST is executed on a separate `Thread`. All nodes inherit the *execute* method from the abstract `Node` class, as shown in Listing 4.9.

```
 1 public abstract class Node {
 2   ...
 3   public void execute(Interpreter interpreter) throws ControlFlowException {
 4     if (interpreter.isDebug()) {
 5       interpreter.setLineOfExecution(line);
 6       Object lock = interpreter.getLock();
 7       synchronized (lock) {
 8         try {
 9           isBreakpoint = true;
10           AbstractSyntaxTree.writeASTToFile(interpreter.getRoot());
11           lock.wait();
12         }
13         catch (InterruptedException e) {
14           throw new RuntimeException(e);
15         }
16         finally {
17           isBreakpoint = false;
18         }
19       }
20     }
21   }
22   ...
23 }
```

Listing 4.9: Single step implementation for all nodes

All nodes have a field *line*, which links the node to its corresponding source code statement. In line 5, the line of execution is set. This information is necessary for the `ListView` element to highlight the current line in the source code. In line 7, the execution `Thread` synchronizes on the shared *lock* object. In line 9 the current node marks itself so that the DOT string of the current node is highlighted, as well as the `TreeView` element. To display the highlighted AST graph, the AST is updated in line 10. In line 11, the execution `Thread` puts itself to sleep until the `Step` button wakes the `Thread` by using the shared *lock* object.

Chapter 5

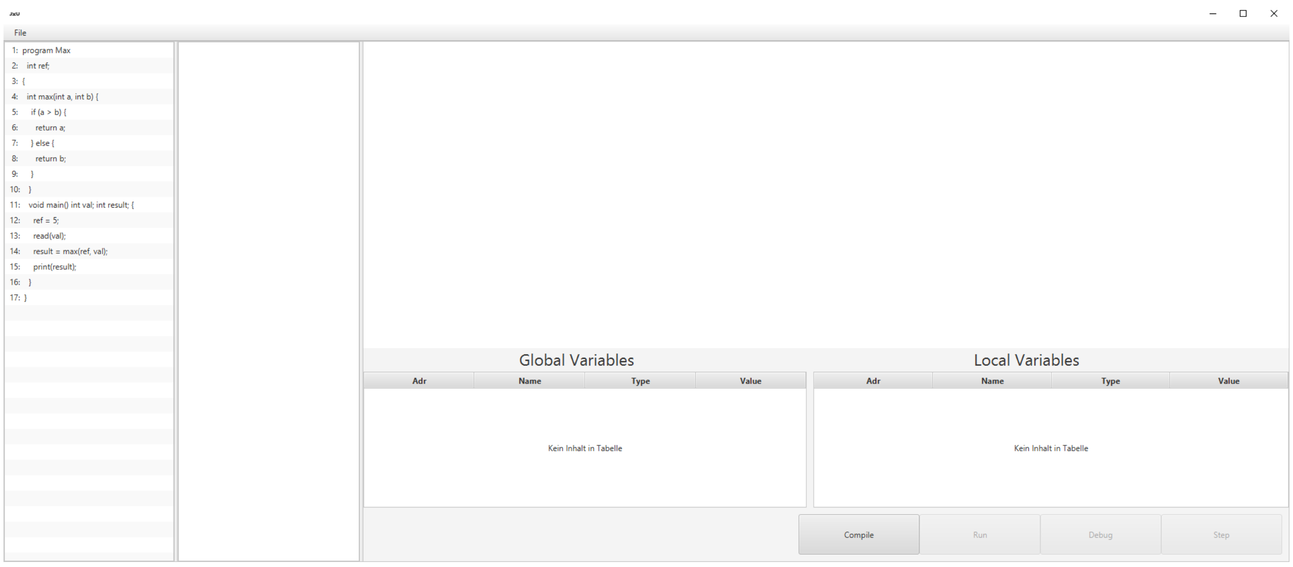
Demo

In this section, a brief demonstration of the AST interpreter implementation is presented. The example program in Listing 5.1 implements the *max* function in MicroJava, which returns the larger value of two variables.

```
program Max
  int ref;
{
  int max(int a, int b) {
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }
  void main() int val; int result; {
    ref = 5;
    read(val);
    result = max(ref, val);
    print(result);
  }
}
```

Listing 5.1: Demo code of *max* function in MicroJava

After starting the application, a `.mj` file can be selected by clicking on the `File` button in the top-left corner. A `FileChooser` is used to navigate and select the desired file. When the MicroJava program is open, the source code is displayed on the left side of the interface. The only available action is pressing the `Compile` button, as shown in Figure 5.1.



```
program Max
 int ref;
{
 int max(int a, int b) {
  if (a > b) {
   return a;
  } else {
   return b;
  }
 }
 void main() int val; int result; {
  ref = 5;
  read(val);
  result = max(ref, val);
  print(result);
 }
}
```

Global Variables

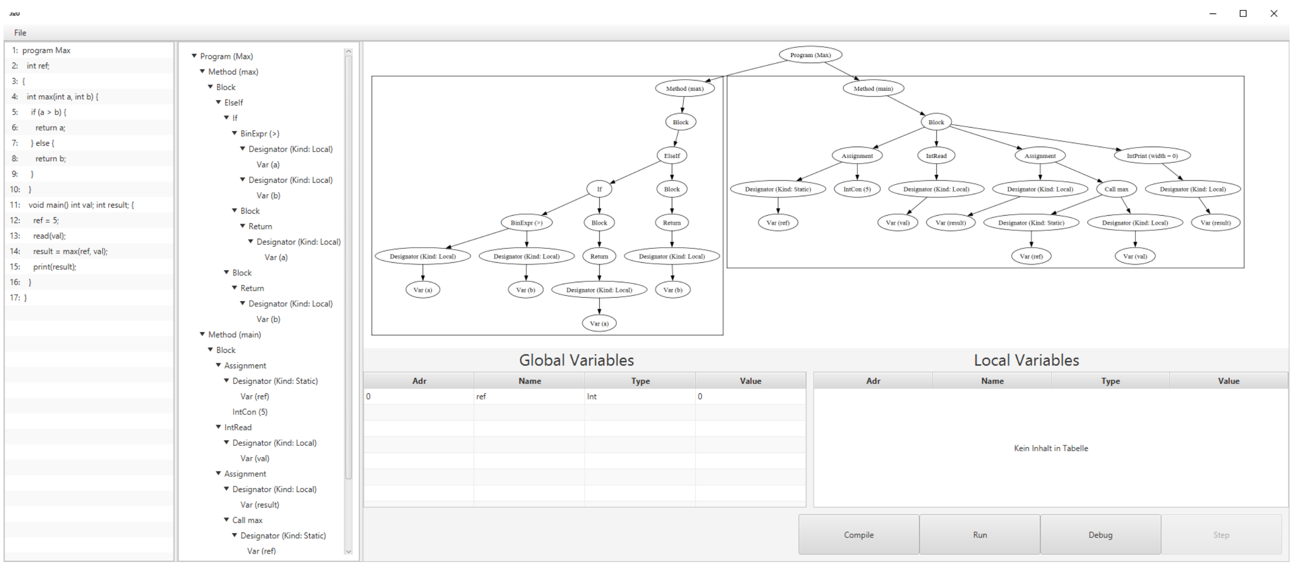
Adr	Name	Type	Value
Kein Inhalt in Tabelle			

Local Variables

Adr	Name	Type	Value
Kein Inhalt in Tabelle			

Figure 5.1: Interface after loading `max.mj`

Only after successful compilation, the source code is visualized as an AST, both as a `TreeView` and in a graph representation, as shown in Figure 5.2. The program can then be executed by pressing the `Run` button. The output is displayed in Figure 5.3



```
program Max
 int ref;
{
 int max(int a, int b) {
  if (a > b) {
   return a;
  } else {
   return b;
  }
 }
 void main() int val; int result; {
  ref = 5;
  read(val);
  result = max(ref, val);
  print(result);
 }
}
```

```
▼ Program (Max)
  ▼ Method (max)
    ▼ Block
      ▼ ElseIf
        ▼ If
          ▼ BinExpr (>)
            ▼ Designator (Kind: Local)
                Var (a)
            ▼ Designator (Kind: Local)
                Var (b)
          ▼ Block
            ▼ Return
              ▼ Designator (Kind: Local)
                  Var (a)
        ▼ Block
          ▼ Return
            ▼ Designator (Kind: Local)
                Var (b)
  ▼ Method (main)
    ▼ Block
      ▼ Assignment
        ▼ Designator (Kind: Static)
            Var (ref)
          IntCon (5)
      ▼ IntRead
        ▼ Designator (Kind: Local)
            Var (val)
      ▼ Assignment
        ▼ Designator (Kind: Local)
            Var (result)
        ▼ Call max
          ▼ Designator (Kind: Static)
              Var (ref)
```

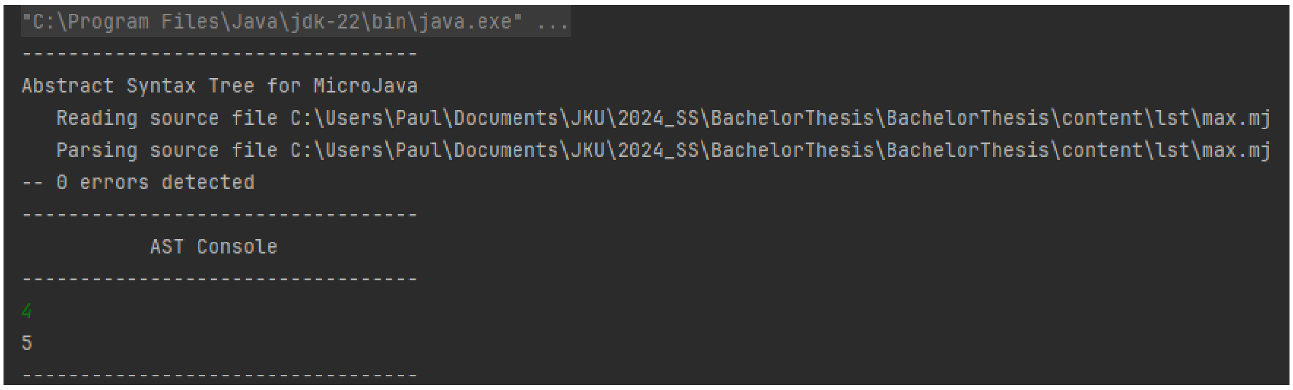
Global Variables

Adr	Name	Type	Value
0	ref	Int	0

Local Variables

Adr	Name	Type	Value
Kein Inhalt in Tabelle			

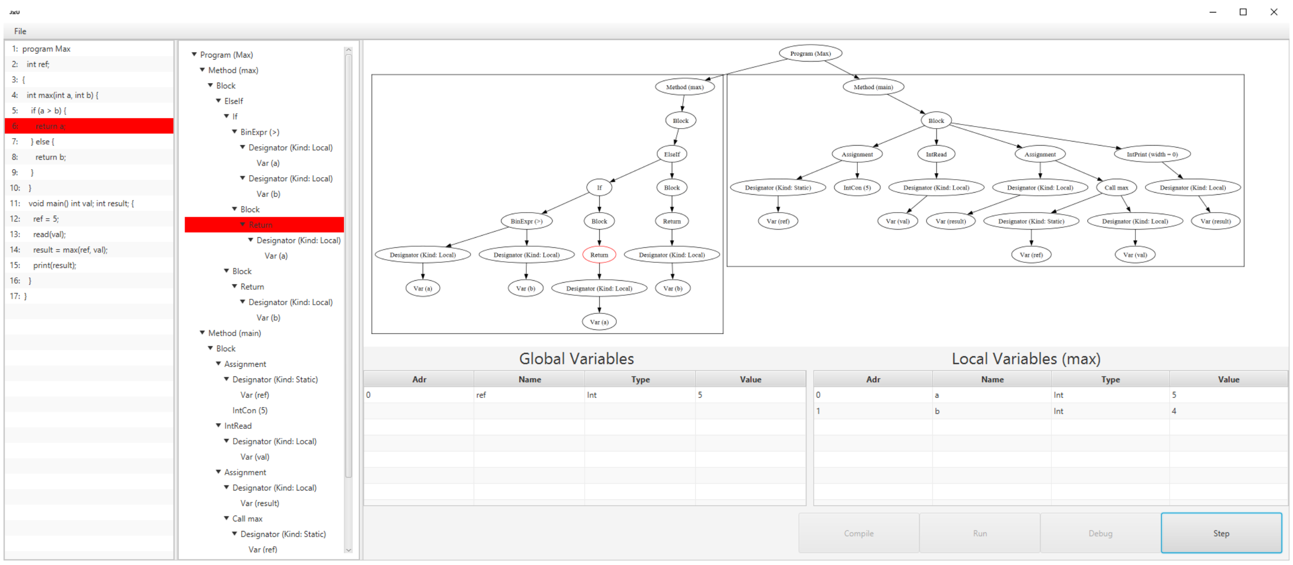
Figure 5.2: Interface after compiling `max.mj`



```
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
---------------------------------
Abstract Syntax Tree for MicroJava
   Reading source file C:\Users\Paul\Documents\JKU\2024_SS\BachelorThesis\BachelorThesis\content\lst\max.mj
   Parsing source file C:\Users\Paul\Documents\JKU\2024_SS\BachelorThesis\BachelorThesis\content\lst\max.mj
-- 0 errors detected
---------------------------------
           AST Console
---------------------------------
4
5
---------------------------------
```

Figure 5.3: Console output of `max.mj` with `val = 4`

Optionally, the program can be executed in debug mode by pressing the `Debug` button. In this mode, the AST interprets the code node by node, with each step initiated by pressing the `Step` button. In each step the current code line, `TreeView` item and graph node are highlighted, as shown in Figure 5.4. Additionally, the runtime values of both global and local variables can be observed in the two `TableView` elements. The complete graph representation of `max.mj` is shown in Figure 5.5.
To generate the graph file, the implementation runs a console command that converts the DOT string stored in a `.dot` file into an `.svg` file using the Graphviz engine. For this process, the Graphviz engine must be installed on the local machine and can be downloaded from the following source [15].



```
program Max
  int ref;
{
  int max(int a, int b) {
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }
  void main() int val; int result; {
    ref = 5;
    read(val);
    result = max(ref, val);
    print(result);
  }
}
```

```
▼ Program (Max)
  ▼ Method (max)
    ▼ Block
      ▼ ElseIf
        ▼ If
          ▼ BinExpr (>)
            ▼ Designator (Kind: Local)
                Var (a)
            ▼ Designator (Kind: Local)
                Var (b)
          ▼ Block
            ▼ Return
              ▼ Designator (Kind: Local)
                  Var (a)
        ▼ Block
          ▼ Return
            ▼ Designator (Kind: Local)
                Var (b)
  ▼ Method (main)
    ▼ Block
      ▼ Assignment
        ▼ Designator (Kind: Static)
            Var (ref)
          IntCon (5)
      ▼ IntRead
        ▼ Designator (Kind: Local)
            Var (val)
      ▼ Assignment
        ▼ Designator (Kind: Local)
            Var (result)
        ▼ Call max
          ▼ Designator (Kind: Static)
              Var (ref)
```

Global Variables

Adr	Name	Type	Value
0	ref	Int	5

Local Variables (max)

Adr	Name	Type	Value
0	a	Int	5
1	b	Int	4

Figure 5.4: Program state at `Return` node of *max* function

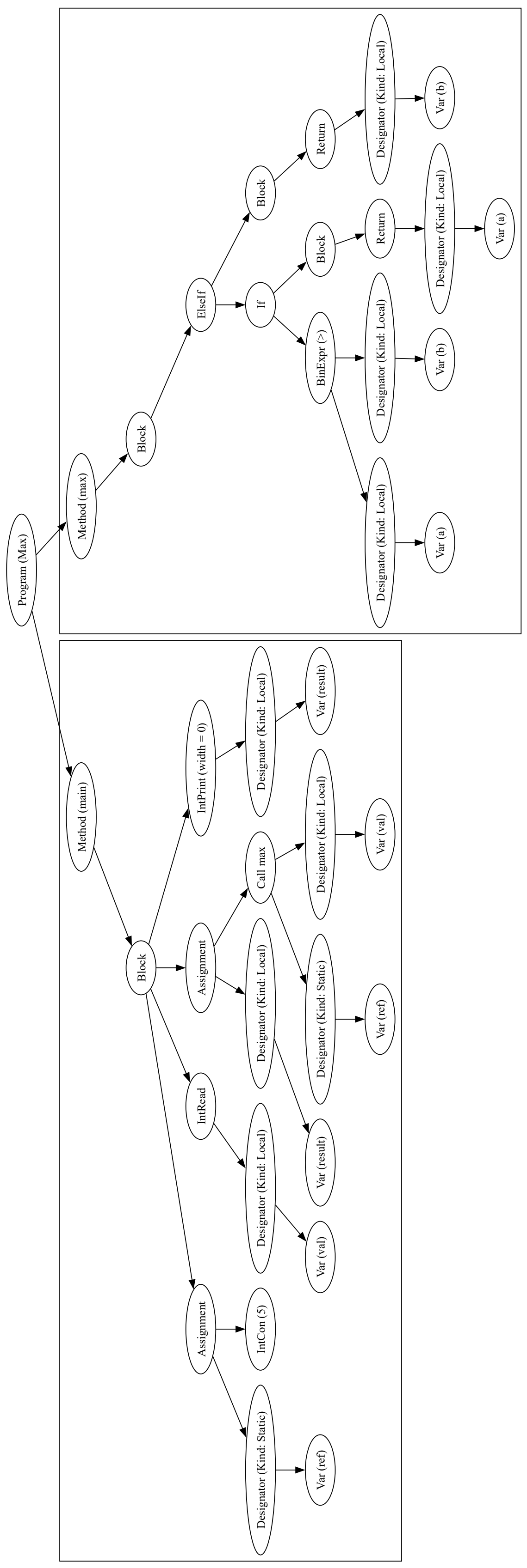


Figure 5.5: Abstract syntax tree of `max.mj`

Chapter 6

Conclusion and Future Work

In this thesis, an AST interpreter was implemented for the programming language MicroJava. The language is defined as an EBNF grammar along with a semantic specification. Using Co-Co/R and an attributed grammar, the scanner and parser classes were generated. Through the attributed grammar, the parser constructs an abstract syntax tree. This tree can be interpreted by traversing it and executing the *execute* method of each node. Traversal of the tree also allows for the generation of a string representation and a `TreeView`. Both representations are utilized to visualize the AST in a JavaFX application. Additionally, a debug mode was implemented to traverse the tree step by step. In this mode, the current code segment is highlighted in all source code representations. The current state of both global and local variables can also be observed. The implementation described by this thesis is available on GitHub [16].

For further optimization, based on the existing project, one could implement constant expression reduction, where numerical expressions are simplified to a single `Con` node during compilation. Additionally, enhancing interactivity could involve transitioning to JavaScript. Utilizing web compatibility, JavaScript could interact directly with the `.svg` file of the AST, rather than generating a new file each time. This approach could also facilitate the creation of an interactive graph, where nodes respond to user interactions, such as expanding to show child nodes or displaying internal values.

List of Figures

Listings

Bibliography

[1] Hanspeter Mössenböck. *Sprechen Sie Java?* dpunkt.verlag, 2014.

[2] Hanspeter Mössenböck. *Compilerbau - Grundlagen und Anwendungen.* dpunkt.verlag, 2024.

[3] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we?, 2023.

[4] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools.* pearson Education, 2007.

[5] The compiler generator coco/r. `https://ssw.jku.at/Research/Projects/Coco/`. Accessed: 2024-06-29.

[6] 04.semantikanschluss - unterlagen zur vorlesung compilerbau. `https://ssw.jku.at/Teaching/Lectures/CB/VL/`. Accessed: 2024-08-12.

[7] The compiler generator coco/r - user manual. `https://ssw.jku.at/Research/Projects/Coco/Doc/UserManual.pdf`. Accessed: 2024-06-29.

[8] graphviz documentation. `https://graphviz.org/doc/info/lang.html`. Accessed: 2024-08-13.

[9] online dot language visualisation. `https://dreampuf.github.io/GraphvizOnline`. Accessed: 2024-08-13.

[10] Javafx documentation. `https://openjfx.io`. Accessed: 2024-08-14.

[11] Scene builder. `https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html`. Accessed: 2024-08-30.

[12] Richard Feynman. Ebnf: A notation to describe syntax. 2016.

[13] 00.unterlagen - unterlagen zur vorlesung compilerbau. `https://ssw.jku.at/Teaching/Lectures/CB/VL/`. Accessed: 2024-08-12.

[14] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 13–14, New York, NY, USA, 2012. Association for Computing Machinery.

[15] graphviz engine download. `https://graphviz.org/download/`. Accessed: 2024-09-18.

[16] MicroJava ast interpreter. `https://github.com/SSW-JKU/microjava-ast-interpreter`. Accessed: 2024-09-23.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 27. September 2024

Paul Lehner

Signed by:	Paul Lehner
Date:	27.09.2024 18:03:08

MicroJava – Grammar 2023

```
Program        = "program" ident { ConstDecl | VarDecl | ClassDecl }
                 "{" {MethodDecl} "}".


ConstDecl      = "final" Type ident "=" ( number | charConst ) ";".

VarDecl        = Type ident { "," ident } ";".

ClassDecl      = "class" ident "{" { VarDecl  } "}".

MethodDecl     = ( Type | "void" ) ident "(" [ FormPars ] ")"
                 { VarDecl } Block.

FormPars       = Type ident { "," Type ident }.

Type           = ident [ "[" "]" ].


Block          = "{" { Statement } "}".

Statement      = Designator ( Assignop Expr | ActPars | "++" | "--" ) ";"
               | "if" "(" Condition ")" Statement [ "else" Statement ]
               | "while" "(" Condition ")" Statement
               | "break" ";"
               | "return" [ Expr ] ";"
               | "read" "(" Designator ")" ";"
               | "print" "(" Expr [ "," number ] ")" ";"
               | Block
               | ";".

Assignop       = "=" | "+=" | "-=" | "*=" | "/=" | "%=".

ActPars        = "(" [ Expr { "," Expr } ] ")".


Condition      = CondTerm { "||" CondTerm }.

CondTerm       = CondFact { "&&" CondFact }.

CondFact       = Expr Relop Expr.

Relop          = "==" | "!=" | ">" | ">=" | "<" | "<=".


Expr           = [ "-" ] Term { Addop Term }.

Term           = Factor { Mulop Factor | "**" number }.

Factor         = Designator [ ActPars ]
               | number
               | charConst
               | "new" ident [ "[" Expr "]" ]
               | "(" Expr ")".

Designator     = ident { "." ident | "[" Expr "]" }.

Addop          = "+" | "-".

Mulop          = "*" | "/" | "%".
```