The Mini-Java Programming Language

Jackson Rauup[*] Gustavo Bezerra[†] Vitor Greati[‡] Joel Felipe [§]

October 31, 2018

Contents

[*]jacksonrauup@hotmail.com

[†]gustavowl@lcc.ufrn.br

[‡]vitorgreati@gmail.com

[§]joelfelipe07@gmail.com

Participation

The table below expresses the percentage of participation of each member in the realization of the versions of this work.

Version 1

Name	Participation (%)
Jackson Rauup	25
Joel Felipe	25
Gustavo Alves	25
Vitor Greati	25

Table 1: Participation in Version 1.

Version 2

Name	Participation (%)
Jackson Rauup	25
Joel Felipe	25
Gustavo Alves	25
Vitor Greati	25

Table 2: Participation in Version 2.

Version 3

Name	Participation (%)
Jackson Rauup	25
Joel Felipe	25
Gustavo Alves	25
Vitor Greati	25

Table 3: Participation in Version 3.

Version 4

Name	Participation (%)
Jackson Rauup	25
Joel Felipe	25
Gustavo Alves	25
Vitor Greati	25

Table 4: Participation in Version 4.

1 Mini-Java Language Reference

1.1 Syntax

The following sections are dedicated to illustrate examples of valid Mini-Java constructs and other syntactic aspects.

1.1.1 Reserved words

The words listed below have special usages in Mini-Java, so that they cannot be used as identifiers. Mini-Java is case sensitive, so such words are reserved just for their lowercase versions.

```
program
class
method
val
int
string
void
declarations
enddeclarations
if
else
for
to
step
while
switch
case
print
read
not
```

1.1.2 Valid Identifiers

Identifiers are used for uniquely representing variables, classes, programs, methods and parameters. Every identifier accepts alphanumeric characters plus the underline ("_"). The ids' first character cannot be a number, only a letter or an underline. The following examples are valid identifiers, except for the last one:

```
abcd2
a1b2c3
AaBb
black_sabbath
_
___
_abcd
_AbCd
_1234
```

2abcd

1.1.3 Data Types

The Mini-Java language supports the following types: integer, string, arrays, and personalized ones, given by classes, in the object-oriented approach. The integer type is denoted by the keyword `int`, the string type is denoted by `string`, and the array type is represented by some type name followed by a sequence of left and right square brackets. Integer literals are just written as ordinary numbers and string constants are written between double quotes. Classes are explained in Section 1.1.11.

1.1.4 Expressions

There are three types of expressions: arithmetic, logical and relational. Arithmetic expressions are those which result in integers or strings, while logical and relational expressions have boolean-valued outcomes. In therms of syntax, the three expressions may be intertwined; though this may be semantically invalid.

The arithmetic expressions mix the basic operations, literal constants, variables, more expressions and even method calls. The following are examples of valid arithmetic expressions. Note that the expressions are not associated to a context, e.g. variable assignments, since they may appear in many different situations.

```
1
+5
-7
(17)
90 + 90
-21 + 12
90 * 2
78 / 3
25 % 4
add(2, 5)
add(num, 5*7)
assume("control", 2112)
doMagic(i * (2 + const) - shift, (flag * 5) + year / month)

#These expressions are not standalone.
#They may be part of variable assignments or relational expressions
```

The remaining expressions are even more heterogeneous than arithmetic expressions. Logic and Relational expressions may combine arithmetic expressions with boolean operations (*logical and*, and *logical or*) and relations (less than, equals, different, greater or equal, etc.).

```
73
73 == 1001001
2112 < 5
isEven(number) && isPrime(number) && number == 2
(1 || false) && true
length("Rush") <= length("Caress of Steel") - 13

```

```
8 #Relational statements are normally encapsulated in statements
9 #that may change the execution flow, e.g. if, while and for
```

It may seem confusing to treat integers as valid logic expressions. However, note that Mini-Java does not natively support boolean variables. Henceforth, similarly to C, integers will be used to simulate booleans. Also, note that `false` and `true` in line 5 are actually variables; hence, the expression may be evaluated to false if `true == 0`.

1.1.5 Variables Declaration and Initialization

Declarations can only be stated in the beginning of the class body. It must contain a type (Section 1.1.3), a valid variable identifier and an optional initialization value. Many variables of the same type may be declared in the same line, separated by commas. Every declaration must finish with a semicolon.

A valid variable identifier consists of an identifier (Section 1.1.2) followed by optional brackets. The brackets represent arrays. For instance, the following example illustrates valid identifiers (the first two) and invalid variable identifiers (last two):

```
color
names[]
[]
telephoneNumbers[][]
```

The initial value may be either expressions (Section 1.1.4) or array initializers or creators. The array creator will be responsible for explicitly specifying the size of the array, even though it does not specify its contents. On the other hand, the array initializer implicitly declares the array size by stating its value.

Combining the variables identifiers with the expressions and arrays initializers and creators, it is possible to obtain the following valid examples:

```
string str;
int a, b = 2, c, d, e = 5;
string[] str_arr = @string[10];
int[] vec = @int [size + 2];
int[][] identity = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
int[][] unknown_line = { @int[2], {73, 2112} };
#this shall be generally inside a "declarations" block
```

1.1.6 Assignments

Similar to Pascal, the assignment operator is `:=`, which can't be used to initialize a variable, only for assignment. The examples below show how assignments are written in Mini-Java:

```
#this shall be inside a method definition (class body)
declarations
    string str;
    int a, b , c = 8;
    string[] str_arr = @string[10];
enddeclarations

```

```
str := "text";
a := 2;
b := 3;
c := 5;
str_arr[a] := "a";
str_arr[5] := "c";
```

1.1.7 Conditional structures

The project language supports the following conditional structures: If/Else and Switch/-Case. The following codes exemplify their use:

1.1.7.1 If/Else Statements

```
#this shall be inside a method definition (class body)
declarations
    string str;
    int a = 5;
enddeclarations

if a > 10 {
    str := "bigger"
} else if a == a {
    str := "equal"
} else {
    str := "smaller"
}
```

1.1.7.2 Switch/Case Statements

```
#this shall be inside a method definition (class body)
declarations
    string[] str_arr = @string[10];
    int control = 3;
enddeclarations

switch(control) {
    case 0 { str_arr[0] := "zero" }
    case 1 { str_arr[0] := "one" }
    case 2 { str_arr[0] := "two" }
    case 3 { str_arr[0] := "three" }
    case 4 { str_arr[0] := "four" }
    default { str_arr[0] := "minus one"}
}
```

1.1.8 Repetition structures

The repetition structures available in the language project are "while" and "for" statements. The structure of while is similar to that of Java, with the difference that parentheses are optional. The for statement is similar to that of Pascal, using only one initialization, one expression for the limit of the variable loop and allowing to define a step via the `step` keyword. Examples of these constructs are below.

1.1.8.1 While Statement

```
#this shall be inside a method definition (class body)
declarations
    int i = 0;
    int max = 100;
enddeclarations

while i < 100 {
    i := i + 1;
}
```

1.1.8.2 For Statement

```
#this shall be inside a method definition (class body)
declarations
    int max = 100;
    int i;
enddeclarations

for i := 0 to max {
    i := i + 1;
}

for i := max to 10 step -1 {
    i := i - 1;
}
```

1.1.9 Subprograms declaration and call

All subprograms need to be declared with the keyword `method`, followed by a return type (use `void` for no return), an identifier, a list of formal parameters between parentheses, and, finally, the statements block. Such block can begin with a section of variable declarations. To call a subprogram, one just have to type its name and provide, between parentheses, the list of actual parameters. Below is an example of a class with various subprograms:

```
program MethodExamples;

class Math {

    declarations
        int i = 1;
    enddeclarations

    method void toIncrease() {
        i := i + 1
    }

```

```
    method void toDecrease() {
        i := i - 1
    }

    method void toIncreaseN(int n) {
        int j = 0;
        while (j < n) {
            toIncrease();
            j := j + 1
        }

    }

    method void toDecreaseN(int n) {
        int j = 0;
        while (j < n) {
            toDecrease();
            j := j + 1
        }
    }

    method int toIncrease() {
        i := i + 1;
        return i
    }

    method int toDecreaseN(int n) {
        i := i - n;
        return i
    }
}
```

1.1.10 Statement blocks

Blocks are statements lists containing optionally a variables declarations section. Below, an example of one that declares its local variables:

```
declarations
    String str;
    int a, b , c = 8;
enddeclarations

{
    str := "text";
    a := 2;
    b := 3;
    c := 5;
}
```

Next, an example of one without the declaration section:

```
{
```

```
    str := "text";
    a := 2;
    b := 3;
    c := 5;
}
```

Blocks are commonly used in the definition of methods, specifying the list of statements to be executed, as well as the local variables to be used.

1.1.11 Classes

A class is divided in two parts: declaration, which assigns the name of the class; and body, which describes its variables declarations and methods.

The class body necessarily describes all declarations before the methods. The body is allowed to have neither declaration or methods. The following examples illustrate valid classes declarations:

```
class ClassExample1
{
    declarations
        int i = 0;
    enddeclarations

    method void method1() {}
}

class ClassExample2
{
    method void method1() {}
}

class ClassExample3
{
    declarations
        int i = 0;
    enddeclarations
}

class ClassExample4
{
}
```

1.1.12 IO

Similar to many languages, the IO's commands are "print" and "read". They print an expression to and read an identifier from the standard IO. The following examples illustrate valid IO's commands usages:

```
#this code shall be inside a body of a class or method
print "Hello World"
print a + b
```

```
print "Hello " + world
read world
```

1.1.13 Comments

Comments can be made using "#" and all of characters after this symbol will be ignored. The following examples illustrate valid comments:

```
#This line is commented
a := "This part is not commented" #This part is commented
```

2 Examples

This section presents three very important algorithms written in Mini-Java to exemplify its syntax.

2.1 Merge Sort

```
program MergeSortProgram;
class MergeSort
{
    method void merge(int[] v; int begin; int middle; int end)
        declarations
            int i = begin, j = middle+1, k = begin, length = end-begin+1;
            int[] aux;
        enddeclarations
    {
        while (i <= middle) && (j <= end)
        {
            if v[i] < v[j] {
                aux[k] := v[i];
                i := i + 1
            }
            else
            {
                aux[k] := v[j];
                j := j + 1
            };
            k := k + 1
        };

        while i <= middle
        {
            aux[k] := v[i];
            i := i + 1;
            k := k + 1
        };

        while j <= end
        {
            aux[k] := v[j];
            j := j + 1;
            k := k + 1
        };

        for k := begin to end step 1
        {
            aux[k] := v[k]
        }
    }
    method void mergeSort(int[] v; int begin; int end)
```

```
        declarations
            int middle;
        enddeclarations
    {
        if begin < end
        {
            middle := (end + begin)/2;
            mergeSort(v, begin, middle);
            mergeSort(v, middle, end);
            merge(v, begin, middle, end)
        }
    }
}
```

2.2 Binary search

```
program BinarySearchProgram;

class BinarySearch
{
    method int binarySearch(int[] arr; int l; int r; int x)
        declarations
            int mid;
        enddeclarations
    {
        if r >= 1
        {
            mid := l + (r - l)/2;
            if arr[mid] == x
            {
                return mid
            }
            else if arr[mid] > x
            {
                return binarySearch(arr, l, mid-1, x)
            };
            return binarySearch(arr, mid+1, r, x)
        };
        return -1
    }
}
```

2.3 Quick Sort

```
program QuickSortProgram;

class QuickSort
{
    method void quickSort(int[] v; int begin, end)
        declarations
            int i = begin, j = end-1, pivot = v[(begin + end)/2], aux;
```

```
        enddeclarations
    {
        while i <= j
        {
            while (v[i] < pivo) && (i < end)
            {
                i := i + 1
            };
            while (v[j] > pivo) && (j > begin)
            {
                j := j - 1
            };
            if i <= j
            {
                aux := v[i];
                v[i] := v[j];
                v[j] := aux;
                i := i + 1;
                j := j - 1
            }
        };
        if j > begin
        {
            quickSort(v, begin, j + 1)
        };
        if i < end
        {
            quickSort(v, i, end)
        };
        return
    }
}
```

2.4 Specific constructs

```
program Example;

class Useless { }

class QuasiUseless {
    declarations
    enddeclarations
}

class HalfUseless {
    declarations
        MyClass[] v = @MyClass[10];
        string a, b;
        int[][] identity = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
    enddeclarations
}
```

```

class Switch {
    method string map(val int key) {
        switch key {
            case 1 {
                return "A"
            }
            case 2 {
                return "B"
            }
            default {
                return "Invalid"
            }
        };
        print "Finished."
    }
}

class Expr {
    declarations
        MyClass[][] v = @MyClass[10][10];
        int a, b;
        int bool;
    enddeclarations
    method void makeExpr() {
        read a;
        bool := not ((a != b) && b || (a || b));
        bool := a + b - a * b / a % b + a + v[2,3].name[19];
        a := -2;
        a := +2;
        makeExpr()
    }
}
```

2.5 Program with errors

```
program string ;

class $oi {
    declarations
        int a = /2;
    enddeclarations
}
```

3 Extended Backus-Naur Form (EBNF)

⟨program⟩ ⊨ "program" ⟨id⟩ ";" ⟨class-decl⟩{⟨class-decl⟩}
⟨class-decl⟩ ⊨ "class" ⟨id⟩ ⟨class-body⟩
⟨class-body⟩ ⊨ "{" [⟨decls⟩] {⟨method-decl⟩} "}"
⟨decls⟩ ⊨ "declarations" {⟨field-decl⟩ ";"} "enddeclarations"
⟨field-decl⟩ ⊨ ⟨type⟩ ⟨field-decl-aux⟩
⟨field-decl-aux⟩ ⊨ ⟨var-decl-id⟩ ["," ⟨field-decl-aux⟩] | ⟨var-decl-id⟩ "=" ⟨var-init⟩ ["," ⟨field-decl-aux⟩]
⟨var-decl-id⟩ ⊨ ⟨id⟩ {"[" "]"}
⟨var-init⟩ ⊨ ⟨expression⟩ | ⟨array-init⟩ | ⟨array-creation-expr⟩
⟨array-init⟩ ⊨ "{" ⟨var-init⟩ {"," ⟨var-init⟩} "}"
⟨array-creation-expr⟩ ⊨ "@"⟨type⟩ ⟨array-dim-decl⟩ {⟨array-dim-decl⟩}
⟨array-dim-decl⟩ ⊨ "[" ⟨expression⟩ "]"
⟨method-decl⟩ ⊨ "method" ⟨method-decl-aux⟩ ⟨id⟩ "(" [⟨formal-param-list⟩] ")" ⟨block⟩
⟨method-decl-aux⟩ ⊨ "void" | ⟨type⟩
⟨formal-param-list⟩ ⊨ ["val"] ⟨type⟩ ⟨id⟩ {"," ⟨id⟩} [";" ⟨formal-param-list⟩]
⟨block⟩ ⊨ [⟨decls⟩] ⟨stmt-list⟩
⟨type⟩ ⊨ ⟨type-aux⟩ {"[]"}
⟨type-aux⟩ ⊨ ⟨id⟩ | "int" | "string"
⟨stmt-list⟩ ⊨ "{" ⟨stmt⟩ {";" ⟨stmt⟩} "}"
⟨stmt⟩ ⊨ ⟨assign-stmt⟩ | ⟨method-call-stmt⟩ | ⟨return-stmt⟩ | ⟨if-stmt⟩
⟨stmt⟩ ⊨ ⟨while-stmt⟩ | ⟨for-stmt⟩ | ⟨switch-stmt⟩ | ⟨print-stmt⟩ | ⟨read-stmt⟩
⟨assign-stmt⟩ ⊨ ⟨variable⟩ ":=" ⟨expression⟩
⟨method-call-stmt⟩ ⊨ ⟨variable⟩ "(" [⟨expression⟩ {"," ⟨expression⟩}] ")"
⟨return-stmt⟩ ⊨ "return" [⟨expression⟩]
⟨if-stmt⟩ ⊨ "if" ⟨expression⟩ ⟨stmt-list⟩ ["else" ⟨if-stmt-aux⟩]
⟨if-stmt-aux⟩ ⊨ ⟨if-stmt⟩ | ⟨stmt-list⟩
⟨for-stmt⟩ ⊨ "for" ⟨assign-stmt⟩ "to" ⟨expression⟩ ["step" ⟨expression⟩] ⟨stmt-list⟩
⟨while-stmt⟩ ⊨ "while" ⟨expression⟩ ⟨stmt-list⟩
⟨switch-stmt⟩ ⊨ "switch" ⟨expression⟩ "{" ⟨case⟩ {⟨case⟩} [⟨default-stmt⟩] "}"
⟨case⟩ ⊨ "case" ⟨expression⟩ ⟨stmt-list⟩
⟨default-stmt⟩ ⊨ "default"⟨stmt-list⟩
⟨expression⟩ ⊨ ⟨simple-expr⟩ [⟨rel-op⟩ ⟨simple-expr⟩]
⟨rel-op⟩ ⊨ "<" | "<=" | "==" | "!=" | ">=" | ">"
⟨simple-expr⟩ ⊨ ⟨term⟩ {⟨simple-bin-op⟩⟨term⟩}
⟨simple-un-op⟩ ⊨ "+" | "-"
⟨simple-bin-op⟩ ⊨ "+" | "-" | "||"
⟨term⟩ ⊨ ⟨factor⟩ {⟨term-bin-op⟩⟨factor⟩}
⟨term-bin-op⟩ ⊨ "*" | "/" | "&&" | "%"
⟨factor⟩ ⊨ ⟨unsig-lit⟩ | ⟨variable⟩ | ⟨method-call-stmt⟩ | "(" ⟨expression⟩ ")" | "not" ⟨factor⟩

$$
\begin{aligned}
\langle\text{factor}\rangle &\models \langle\text{simple-un-op}\rangle\langle\text{factor}\rangle \\
\langle\text{unsig-lit}\rangle &\models \langle\text{integer-lit}\rangle \;\mid\; \langle\text{string-lit}\rangle \\
\langle\text{variable}\rangle &\models \langle\text{id}\rangle\ \{\langle\text{variable-aux}\rangle\} \\
\langle\text{variable-aux}\rangle &\models \texttt{"["}\ \langle\text{expression}\rangle\ \{\texttt{","}\ \langle\text{expression}\rangle\}\ \texttt{"]"} \;\mid\; \texttt{"."}\ \langle\text{id}\rangle \\
\langle\text{print-stmt}\rangle &\models \texttt{"print"}\ \langle\text{expression}\rangle \\
\langle\text{read-stmt}\rangle &\models \texttt{"read"}\ \langle\text{id}\rangle
\end{aligned}
$$

4 mjc: a compiler for Mini-Java

4.1 Compiler steps

A compiler is generally organized in two big parts: the front-end and the back-end. Each of them have sub-parts, which will be detailed in the course of this section.

Also called the *analysis* part of the compiler, the first step worries about precisely recognizing and representing the form of the source program in a precise and meaningful way. The basic tasks are to identify each language symbol in the code – which can be a string of one or more characters – and submit the resulting sequence of symbols to the set of grammar rules that characterize the language syntax. The result of this process is an intermediate representation of the program that will be used in the next steps. Some optimization may occur as one or more interleaved step in this process, but *mjc* doesn't perform this for a matter of simplicity.

The front-end encompasses two main phases: the *lexical analysis* or *scanning* and the *syntax analysis* or *parsing*. Sections 4.2 and 4.3 detail each one's peculiarities and present how they are implemented in *mjc*.

The back-end phase is about producing semantics for the program. This will be discussed in future versions of this document, since the current focus is in the front-end phase.

4.2 Lexical Analysis or Scanning

The scanning phase is performed by the *lexical analyser*, which reads the stream of characters and groups them into meaningful sequences, named *lexemes*. Each lexeme l is categorized in a *token* t, and the output of the analyser for each lexeme is the token symbol together with an *attribute value* a, say $\langle t, a\rangle$, where a points to an entry in the *symbol table* for t.

Symbol tables are used by compilers to store additional information about the recognized constructs, being incremented during the compilation phases. In the case of variables, for example, it holds the name, the type, the position in storage and any other variable's relevant attribute.

Once a token is recognized, it is transmitted to the parsing phase. Generally, the lexical analyzer is not the compiler's entry point. Instead, the parser – which acts in the parsing phase – occupies such position and calls the lexical analyzer to recognize and retrieve the next token.

There are tools to specify and generate lexical analyzers. One of the most used is *Lex*, or, more recently, *Flex*, which allows to describe the lexemes of each token by regular expressions. *Lex* specify its own language and has a compiler that generates code for simulating the transition diagrams which represent the indicated string patterns.The list of Mini-Java tokens, together with the corresponding lexemes' regular expressions, can be found in Appendix A.1.

`mjc`'s lexical analyzer was generated by *Lex* due to the ease of implementation and the optimizations provided by such tool. An example of output of the lexical analyzer for the Quick Sort algorithm (Section 2.3) is in Appendix A.2. In order to submit a Mini-Java program to it, the following steps must be executed:

1. If `bin/mjclexer` doesn't exist, execute, from the root directory, `make lexer`.

2. Having a program in a file called `program.mj`, one can execute the following command from the root:

   ```
   ./bin/mjclexer [write_path [output_to_std] ] < /path/to/program.mj
   ```

Notice that it is possible to execute the analyzer without arguments, making it printing a table of tokens in the standard output. Also, one can pass a path to a file in which the table will be printed and, having that, also inform if the table must still be printed in the standard output (the default is always to print). The columns of the printed table represent, respectively, for each recognized token: the line of occurrence in the code, the column of occurrence of the first lexeme character, the lexeme length, the token name and the lexeme itself. When a character or sequence is not recognized by the analyzer, a warning is printed in the standard error output.

Finally, the lexical analyzer accepts a compilation flag called `__EXECUTABLE__` which causes the compilation of the `main` function and other auxiliary print functions. The compilation performed when `make lexer` is executed activates such tag, since it is of interest of such usage to execute the analyzer as an ordinary program and show its result. However, when compiling it for the parsing purpose, no execution is needed, as the parser will mainly need the `yylex` function.

4.3 Syntax Analysis or Parsing

Parsing worries about checking if a sequence of tokens can be generated by the language grammar. Generally, the parser produces an intermediate representation in the form of a tree – explicitly or implicitly – to be processed by the rest of the compiler. Also, parsers are expected to inform to the users the occurrence of any syntax errors by intelligible messages.

There are three main parser categories: universal, top-down and bottom-up. The first encompasses algorithms that can parse any grammar, although they can be very inefficient. The other two imposes some restrictions to the grammars, but are faster and suitable for the most common programming language's constructs.

Next sections present the implementation of common parser algorithms for `mjc`. The most important concepts for understanding them were summarized and the instructions for running each parser in this compiler is explained at the end of the corresponding section.

4.3.1 Top-down parsing

In order to execute a top-down parsing, it is necessary to execute a top-down analysis. The grammar productions will be executed starting from an initial non-terminal and a parse tree will be generated. The parse tree may have terminal and non-terminal symbols. However, only terminal symbols will be leaf nodes.

The nodes of the token tree will be compared and matched with the grammar's productions. If the grammar is ambiguous, there may exist multiple parse trees for a given program, considering left or right derivations.

For top-down parsing, the left-most symbols will be analyzed first. However, even if the grammar is unambiguous, the parsing may execute an infinite loop. For instance, any grammar with left- recursive productions can cause an infinite loop. This happens because the algorithm would not stop generating new derivations.

In addition, it may be the case that two different productions generate the same terminal symbol. When this happens, the parser will have to guess which production generated that terminal. Also, if the parse tree does not match the chosen productions, the parser will have to keep backtracking until it has chosen every possible production before it fails. To avoid the need of "looking forward" and backtracking, it is necessary to change the grammar to LL(1) form.

Basically, a LL(1) grammar parser only needs to check one terminal symbol to decide which production should be chosen. This terminal is the next symbol given by the lexer. Hence, LL(1) grammar parsers do not need to backtrack. LL(2) grammars parsers would need to look to two symbols to decide which one to choose, and so on. It is important to note that grammars that are not LL(1) may be inefficient. As a consequence, compiling a programming language would be unfeasible.

LL(1) grammars do not have left recursions as well. However, a formal definition demands the concepts of FIRST and FOLLOW sets.

4.3.1.1 FIRST Set

Essentially, the $FIRST$ set is the set of all terminal symbols that begin strings derived from a given variable. For instance, consider the following simple grammar:

$$\begin{array}{rcl} \langle \mathrm{S} \rangle & \models & \langle \mathrm{A} \rangle \, \langle \mathrm{B} \rangle \\ \langle \mathrm{A} \rangle & \models & \lambda \mid \mathrm{a}\langle \mathrm{A} \rangle \\ \langle \mathrm{B} \rangle & \models & \mathrm{b} \mid \mathrm{b}\langle \mathrm{B} \rangle \end{array}$$

Then, $FIRST(B) = \{b\}$, $FIRST(A) = \{\lambda, a\}$, $FIRST(S) = \{a, b\}$ (due to the λ-production). The $FIRST$ set for the Mini-Java language can be found in Appendix A.3.

4.3.1.2 FOLLOW Set

Intuitively, the $FOLLOW$ set of a non-terminal A is the set of all terminal symbols that can be found immediately to the right of A in some sentential form. Consider the following grammar as an example:

$$\begin{array}{rcl} \langle \mathrm{S} \rangle & \models & \langle \mathrm{A} \rangle \, \langle \mathrm{B} \rangle \\ \langle \mathrm{A} \rangle & \models & \lambda \mid \mathrm{a}\langle \mathrm{A} \rangle \\ \langle \mathrm{B} \rangle & \models & \lambda \mid \mathrm{b}\langle \mathrm{B} \rangle \end{array}$$

Then, $FOLLOW(S) = FOLLOW(B) = \{\$\}$. Also, $FOLLOW(A) = \{\$, b\}$, since $FIRST(B) = \{\lambda, b\}$. The $FOLLOW$ set for the Mini-Java language can be found in Appendix A.4.

4.3.1.3 LL(1) Grammar

Recall that a LL(1) grammar cannot be ambiguous neither have left-recursive productions. Also, a LL(1) may be able to determine which production was taken by looking only to the next terminal symbol.

The grammar presented in this section generates the same language as the grammar in Section 3. However, the left recursive productions were removed and there are no "ambiguous" productions. A production may be ambiguous if a given non-terminal has two different productions with common elements in their FIRST set.

For instance, consider the grammar (which is not LL(1)):

$$
\begin{array}{rcl}
\langle \text{S} \rangle & \models & \langle \text{A} \rangle \mid \langle \text{B1} \rangle \\
\langle \text{A} \rangle & \models & \text{a}\langle \text{A} \rangle \mid \langle \text{B2} \rangle \\
\langle \text{B2} \rangle & \models & \text{b} \mid \text{b}\ \langle \text{B2} \rangle \\
\langle \text{B1} \rangle & \models & \lambda \mid \text{b}\langle \text{B1} \rangle
\end{array}
$$

The grammar is not LL(1) because it is ambiguous. For example, b is in its language, however, it is impossible to know if b was generated by production $B1$ or $B2$.

The restrictions of LL(1) will only guarantee that a production can be chosen given the next terminal. For instance, consider the following grammar snippet:

$$
\begin{array}{rcl}
\langle \text{S} \rangle & \models & \langle \text{A} \rangle\ \langle \text{B} \rangle \mid \langle \text{C} \rangle \\
\langle \text{A} \rangle & \models & \langle \alpha \rangle \mid \langle \beta \rangle
\end{array}
$$

If the previous grammar is LL(1), then $FIRST(\alpha)$ and $FIRST(\beta)$ must be disjoint sets. Otherwise, the parser may not know which production to choose. A similar thought can be traced regarding $FIRST(A)$ and $FIRST(C)$.

In addition, if β generates λ, it will be necessary to check if $FIRST(\alpha)$ and $FOLLOW(A)$ are disjoint sets. In other words, to guarantee that $FIRST(\alpha)$ and $FIRST(B)$ are disjoint sets. Otherwise, the parser will not know which production between αB and βB (i.e. λB) was taken without backtracking.

Analogously, if β generates λ, it will be necessary to verify that $FIRST(B)$ and $FIRST(C)$ are disjoint sets.

The generated LL(1) grammar for the Mini-Java Language can be found in Appendix A.5.

4.3.1.4 Recursive Predictive Parser

The recursive definition of the Predictive Parse is rather intuitive. It basically consists on matching each non-terminal to a procedure.

If the procedure for non-terminal P is called, verify if the current token is in $FIRST(P)$. If it is, consume that token and calls the procedures corresponding to the right side of the chosen production. If it does not match, then error. The procedure for the start symbol is called first, and this makes the parsing start.

`mjc` comes with a recursive predictive parser, which relies on recursive call for functions representing each non-terminal in the language LL(1) grammar. In order to execute it, follow these steps:

1. If `bin/mjcll1` doesn't exist, execute, from the root directory, `make ll1parser`.
2. Having a program called `program.mj`, execute the following command from the root:
   `./bin/mjcll1 R < /path/to/program.mj`

For the **correct** examples in Section 2, this parser outputs:

```
Parsing with LL(1) recursive parsing.
Parsing finished. No output means no parse errors.
```

Error Handling Two types of errors are handled differently in this parser. When some expected token is not encountered in the input, `mjc` inserts one of the expected tokens and continues the parsing process. When a non-terminal A cannot be consumed because none of the terminals in its first set is in the input, the parser skips the input sequence until some of the elements in the set $FIRST(A) \cup FOLLOW(A)$ is found. For example, for the problematic example in Section 2.5, the output is:

```
Parsing with LL(1) recursive parsing.
[mjc   error] (1,9) parse error: unexpected string, expecting identifier
[mjc    note] (1,9) inserting identifier
[mjc warning] (3,7) unknown character or sequence $
[mjc   error] (5,17) parse error: unexpected /, expecting identifier,{,(,@,+,−,
    not,integer literal,string literal
[mjc   error] (5,18) parse error: unexpected integer literal, expecting ;,,
[mjc    note] (5,18) inserting ;
[mjc   error] (5,19) parse error: unexpected ;, expecting identifier,
    enddeclarations,int,string
[mjc    note] (5,19) inserting enddeclarations
[mjc   error] (6,5) parse error: unexpected enddeclarations, expecting },method
[mjc    note] (6,5) inserting }
[mjc   error] (7,1) parse error: unexpected }, expecting class,eof
[mjc    note] (7,1) inserting eof
Parsing finished. No output means no parse errors.
```

4.3.1.5 Non-Recursive Predictive Parser

The iterative version of the Predictive Parser focuses on simulating the calls made in the recursive algorithm. This is done with the help of another stack for the productions and a table called parse table, which associates productions to its terminal symbols in $FIRST$ (check Appendix A.6 for the Mini-Java parse table). Whenever the production stack matches the symbol in the terminal stack, both are popped. Whenever a production is matched, it is popped from the production stack and the right side of the production is inserted into it.

For instance, consider the following grammar:

$$\begin{array}{lcl} \langle \text{S} \rangle & \models & \langle \text{A} \rangle \mid \text{b}\langle \text{B} \rangle \\ \langle \text{A} \rangle & \models & \lambda \mid \text{a}\langle \text{A} \rangle \\ \langle \text{B} \rangle & \models & \lambda \mid \text{b}\langle \text{B} \rangle \end{array}$$

This grammar will generate the parse table found in Table 5:

	a	b	$
S	A	bB	A
A	aA		λ
B		bB	λ

Table 5: Example of Parse Table

So, the behaviour for generating the empty string can be found in Table 6

Production Stack	Token Stack	Production Matched
S$	$	S $\rightarrow$ A
A$	$	A $\rightarrow$ λ
$	$	SUCCESS

Table 6: Generating the empty String using the example grammar

If trying to generate the string "bbb", the step-by-step process can be found in Table 7

Production Stack	Token Stack	Production Matched
S$	bbb$	S → bB
bB$	bbb$	match and pop
B$	bb$	B → bB
bB$	bb$	match and pop
B $	b$	B → bB
bB$	b$	match and pop
B $	$	B → λ
$	$	SUCCESS

Table 7: Generating String "bbb" using the example grammar

Lastly, the procedure of evaluating a String that is not in the language, e.g. "ab", can be found in Table 8.

Production Stack	Token Stack	Production Matched
S$	ab$	S → A
A$	ab$	A → aA
aA$	ab$	match and pop
A$	b$	ERROR

Table 8: Trying to generate invalid String

`mjc` also comes equipped with a non-recursive predictive parser, which can be executed by following the steps below:

1. If `bin/mjcll1` doesn't exist, execute, from the root directory, `make ll1parser`.
2. Having a program called `program.mj`, execute the following command from the root:
   `./bin/mjcll1 N < /path/to/program.mj`

For the **correct** examples in Section 2, this parser outputs:

```
Parsing with LL(1) non-recursive parsing.
Parsing finished. No output means no parse errors.
```

Error Handling The compiler tries to match the current token with an element of the stack, in order to continue the parser. For that, it verify if the current token and top of stack have a production in parse table. If this is the case,then it pushes the production in stack and continues the parsing, otherwise it pops the stack and verifies again, while the stack is non empty. When a terminal is on top of the stack, but the current symbol doesn't match, an error message is shown and a pop is performed in the stack. If a non-terminal doesn't have an entry in the parse table for the current token, an error is shown and the stack is popped until a non-terminal is found whose predict set contains the current input token. For example, for the problematic example in Section 2.5, the output is:

```
Parsing with LL(1) non-recursive parsing.
[mjc   error] (1,9) parse error: unexpected string, expecting identifier
[mjc   error] (1,9) parse error: unexpected string, expecting ;
[mjc   error] (1,9) parse error: unexpected string, expecting class
Parsing finished. No output means no parse errors.
```

4.3.2 Bottom-up parsing

A bottom-up parse can be seen as the process of reducing a string w to the start symbol of the grammar. It corresponds to the construction of a parse tree starting from the leaves and going up to the root. A common style of bottom-up parse is known as *shift-reduce*, for which the largest class of appropriate grammars are called LR.

A reduction step in this context means the replacement of a specific substring matching the right-hand side by the left-hand side, or read, of a production rule. In other words, it is the reverse of a derivation step. A substring that can be replaced in this way is called a *handle*. Unambiguous grammars always have only one handle for each right-sentential form of their rules.

In shift-reduce parse, a stack holds grammar symbols, and the input buffer keeps the substring that remains to be parsed, in such a way that the handle to be reduced next is always at the top of the stack. Up to reaching the reduce moment, tokens in the input stream are *shifted* (moved) onto the stack.

Reduce and shift are the main operations performed in a shift-reduce parse. Actually, there are four in this context, which can be summarized as:

Shift Move the token in the start of the input onto the top of the stack.

Reduce A string to be reduced in on top of the stack, so decide with what nonterminal to replace it.

Accept Announce successful completion of parsing.

Error Discover a syntax error and call an error handling routine.

For some context-free grammars, conflicts in the shift-reduce parsing can happen. They occur generally when the parser, even knowing the entire stack and the next k symbols in the input, cannot decide whether to shift or to reduce (shift-reduce conflicts), or which reduction to choose between a set of possible reductions (reduction-reduction conflict). Such grammars are not in the LR(k) class of grammars. Conflicts can be solved by adapting the grammar to a more suitable form, when possible, or even giving priority to some of the choices available (to shift or to reduce, for example).

The next subsections present details about the main shift-reduce parsers and the parser generator Yacc used in this work along with the instructions to compile and test the implemented parser.

4.3.2.1 LR parsing

There are three LR Parsers of interest: LR(0), SLR, and LR(1). The language generated by these parsers are proper subsets of each other: $LR(0) \subset SLR \subset LR(1)$. LR basically means that the input tokens will be read from left to right while the grammar derivations will be executed from the rightmost and in reverse (reduced).

LR(0) parser has no look-ahead. In other words, only the current state is considered for shift reducing. Consider a production $A \to BC$. There are three possible states (named items) for the stack: $A \to \cdot BC$, $A \to B \cdot C$, and $A \to BC\cdot$. The dot indicates the top of the stack. For instance, $A \to B \cdot C$ indicates that the stack is ready to shift C, i.e. expecting a string derived by production C. On the other hand, $A \to BC\cdot$ indicates that the stack is ready to reduce states BC to A.

LR(0) is useful because it allows an automaton to be generated. This automaton will keep track of the states on the stack and it will be used for both SLR and LR(1) in similar versions. In order to generate this automaton, two functions are needed: *closure* and *goto*. The *closure* basically expands the production after the dot until a terminal symbol is found, adding each expansion to the automaton state. The *goto* shifts the next symbol and then implies closure. In order to compute the *goto* it is necessary to consider the look-ahead symbol.

Therefore, one may erroneously conclude that the grammar should be LR(1). However, the closure and goto functions, and the look-ahead will be used *only to generate the automaton*, while the automaton per se represents the LR(0) language, which does not require a look-ahead.

SLR SLR (Simple LR) parsers are better than LR(0) parsers because they put fewer reduce actions into their tables, thus providing more power of constructs distinction in the parsing process. Parser construction in this case is almost the same of LR(0), with the difference that reduce actions appear only where indicated by the $FOLLOW$ set.

LR(1) is the most powerful between the three. LR(1) even expresses all LL(1) languages. LR(1) increments the definition of an item. An item is an LR(0) item (production with a dot in any right-side position) together with a terminal symbol (next input symbol).

The process of generating items is similar to the one of LR(0). The difference is that LR(1)'s process is aided by the $FIRST$ set in the *closure* definition.

4.3.2.2 LALR parsing

The process for generating LR(1) grammar items produces many similar states. The main difference between the states is the look-ahead symbol. Therefore, the idea of LALR is to reduce the number of states by merging repetitive ones.

Let $C = \{I_1, \ldots, I_n\}$ be the items of the LR(1) grammar of language, a core of I_i is the set of first components. We can look for C items having the same core, and we may merge it into one. For example, let $I_1 = [S \to b\cdot, a/b]$ and $I_2 = [S \to b\cdot, \$]$, they form a pair, with core $\{S \to b\cdot\}$. The union of I_1 and I_2 is a item I_{12}, consisting of the set of three items represented by $[S \to b\cdot, a/b/\$]$.

The method LALR(*lookahead*-LR) is useful in practice, because the tables obtained by it are considerably smaller than the LR Parsers, besides that the most common syntactic constructions of programming languages can be expressed by a LALR grammar.

4.3.2.3 Yacc

The Yacc ("Yet another compiler-compiler") is a standard parser generator for the Unix operating system. The Yacc program structure is divided into three sections, separated by the symbols "%%". In the first part are the parser declarations. This one includes the types of values used in the parser stack, declarations of tokens (used in the grammar), as well as other odds and ends. The second part contains the grammar rules (with parts of C code) and the third has the programs. The rules section describes the grammar through a set of production rules. A rule structure consists of a name (left-hand side) followed by the ":" operator and a list of symbols and action code (right-hand side) with a ";" operator at the end of the sentence (indicate the final of each rule). Lastly, the third section contains auxiliary functions written in C language. At this section there are the following mandatory functions: main(), yylex() and yyerror().The main() function need to call the yyparse() to parse the input.

Yacc was used in this work to generate a bottom-up parser. Almost no changes were necessary to translate the LL(1) grammar to the LALR form (check Appendix A.7). The only modification performed was the expression grammar simplification, since Yacc deals with precedence and associativity in a rather natural way.

`mjc`'s bottom-up parser can be executed by following the steps below:

1. If `bin/mjclalr` doesn't exist, execute `make lalrparser` from the project root.

2. Having a program called `program.mj`, execute the following command from the root:
   `./bin/mjclalr < /path/to/program.mj`

The success of the parsing occurs when the output is empty. When errors are encountered, they are printed to the user, showing the occurrence location. The error recovery, however, occurs differently when compared to the LL(1) parser. Yacc accepts the introduction of error productions that specify tokens to be sought in order to continue the parsing after an error has been found. This parser for Mini-Java applies such productions for blocks and other enclosing structures.

For example, the parser output for the problematic program in Section 2.5 is:

```
[mjc   error] (1,15) parse error near string, lexeme string
```

4.3.3 Abstract Syntax Tree construction

An Abstract Syntax Tree (AST) expresses a program in terms of abstractions representing the constructions of the language. Details purely related to syntax, like punctuation marks and operator precedence syntactic peculiarity, do not appear, which makes an AST different from an ordinary parse tree.

Mechanisms of syntax-directed translation can be used to construct an AST of a program based on a grammar. Yacc implements some of them, which were used in this work. The next sections present details about this type of translation and how Yacc provides it.

4.3.3.1 Syntax-directed translation

Syntax-directed translation allows code translation during the parsing process. A common approach for that is to define actions and attach attributes to the grammar symbols at each rule, which is a grammar called Syntax-directed Definition. Such actions determine how the attributes of a symbol are computed in terms of the attributes of other symbols. A complementary notation for syntax-directed definitions are syntax-directed translation schemes. They are context-free grammars with program fragments embedded within production bodies.

The evaluation order of the attributes is very important, since the presence of cycles, for example, would prevent the translation from succeeding. Two types of definitions allows for a correct evaluation: S-attributed, which only allow for attributes in a node to be defined in terms of the children of that node (synthesized attributes); and L-attributes, which allows for dependence on the children, but also on the symbols that appear at the left of the symbol being evaluated (inherited attributes).

The AST construction can be performed using these tools by defining a node, as an attribute, for each grammar symbol and attaching to it, via the actions, the nodes corresponding to their children in the production body.

4.3.3.2 Using Yacc for AST construction

Yacc makes an implementation of L-attributed definitions given a grammar written according to its rules. For that, each grammar symbol in a production can be associated to code segments, from left to right, allowing the computation of synthesized and inherited attributes in an order that guarantees success.

In order to build an AST in this work, a set of classes were implemented to represent the nodes. All of them extends from the class `Node`, which implements a fixed `print` method, that, in turn, calls a virtual method called `show`. Every class must implement its version of `show`, in order the correctly exhibit its information. In fact, those classes' constructors represent the abstract language implemented for Mini-Java, which is listed below in a more simplified syntax (here, `<-` indicates inheritance):

```
Id (pos : Pos, id : string) <- Node;
ConstructList (pos : Pos, constructs : deque<Node*>) <- Node;
Expr (pos : Pos) <- Node;
AlExpr (pos : Pos) <- Expr;
ExprParen (pos : Pos, expr : Expr*) <- AlExpr;
RelExpr (pos : Pos, op : RelOp, lhs : AlExpr*, rhs : AlExpr*) <- Expr;
AlBinExpr (pos : Pos, op : AlBinOp, lhs : AlExpr*, rhs : AlExpr*) <- AlExpr;
AlUnExpr (pos : Pos, op : AlUnOp, alexpr : AlExpr*) <- AlExpr;
LitExpr<T> (pos : Pos, val : T) <- AlExpr;
AccessOperation (pos : Pos) <- Node;
BracketAccess (pos : Pos, expressionList : ConstructList*,
        accessOperation : AccessOperation*) <- AccessOperation;
DotAccess (pos : Pos, id : Id*, accessOperation : ConstructList*) <- AccessOperation;
Var (pos : Pos, id : Id*, accessOperation : AccessOperation*) <- AlExpr;
FunctionCallExpr (pos : Pos, var : Var*, actualParams : ConstructList*) <- AlExpr;
Stmt (pos : Pos) <- Node;
AssignStmt (pos : Pos, var : Var*, expr : Expr*) <- Stmt;
FunctionCallStmt (pos : Pos, var : Var*, actualParams : ConstructList*) <- Stmt;
ReadStmt (pos : Pos, id : Id*) <- Stmt;
PrintStmt (pos : Pos, expr : Expr*) <- Stmt;
Case (pos : Pos, expr : Expr*, stmts : ConstructList*) <- Node;
SwitchStmt (pos : Pos, caseList : ConstructList*, defaultStmts : ConstructList*) <- Stmt;
WhileStmt (pos : Pos, expr : Expr*, stmts : ConstructList*) <- Stmt;
ForStmt (pos : Pos, id : Id*, assignExpr : Expr*, toExpr : Expr*,
        stepExpr : Expr*, stmts : ConstructList*) <- Stmt;
ElsePart (pos : Pos) <- Node;
Else (pos : Pos, stmts : ConstructList*) <- ElsePart;
IfStmt (pos : Pos, expr : Expr*, stmts : ConstructList*, elsePart : ElsePart*) <- Stmt;
ElseIf (pos : Pos, ifStmt : IfStmt*) <- ElsePart;
ReturnStmt (pos : Pos, expr : Expr*) <- Stmt;
Type (pos : Pos, numBrackets : int, typeName : string) <- Node;
VarDeclId (pos : Pos, id : Id*, numBrackets : int) <- Node;
VarInit (pos : Pos) <- Node;
FieldDeclVar (pos : Pos, varDeclId : VarDeclId*, varInit : VarInit*) <- Node;
FieldDecl (pos : Pos, type : Type*, varsDecls : ConstructList*) <- Node;
Decls (pos : Pos, fields : ConstructList*) <- Node;
FormalParams (pos : Pos, val : bool, type : Type*, ids : ConstructList*) <- Node;
Block (pos : Pos, decls : Decls*, stmts : ConstructList*) <- Node;
MethodReturnType (pos : Pos, type : Type*) <- Node;
MethodDecl (pos : Pos, returnType : MethodReturnType*, id : Id*,
        params : ConstructList*, block : Block*) <- Node;
ClassBody (pos : Pos, decls : Decls*, methods : ConstructList*) <- Node;
ClassDecl (pos : Pos, id : Id*, body : ClassBody*) <- Node;
Program (pos : Pos, id : Id*, classes : ConstructList*) <- Node;
ExprVarInit (pos : Pos, expr : Expr*) <- VarInit;
ArrayCreationVarInit (pos : Pos, arrayInit : ConstructList*) <- VarInit;
ArrayCreation (pos : Pos, type : Type*, dims : ConstructList*) <- VarInit;
ArrayCreationVarInit (pos : Pos, arrayInit : ArrayCreation*) <- VarInit;
```

Basically, each grammar symbol received a node implementation, which defines the children and other important attributes for the abstract representation. Then, the Yacc grammar implementation was filled with semantic actions in order to connect the nodes during the parse.

In order to test this implementation and check the generated parse tree for a given program, one must follow the same steps for execution of the LALR parser, but calling `make lalrparserast` to compile. In this case, however, when a program is submitted to the parser, the AST is printed in a Javascript-like format. In order to better visualize it, please save the output to a file and use it in the free beautifier available in `https://www.freeformatter.com/javascript-beautifier.html`.

The basic syntax for this notation is:

- A node is expressed by its fields enclosed in curly braces.
- A list of objects is enclosed in square brackets.

Notice that every node carries its position in the source code. For example, consider the following Mini-Java program:

```
program ExampleAST;

class ClassExampleAST {
    declarations
        int a, b = 2;
    enddeclarations

    method string sum(int c) {
        read a;
        read b;
        print a + b + c
    }
}
```

The AST generated by the scheme implemented in Yacc is:

```
{
  pos: [1, 1]
  programName: {
    pos: [1, 9]
    idName: ExampleAST
  }
  classes: {
    pos: [3, 1]
      [{
        pos: [3, 1]
        className: {
          pos: [3, 7]
          idName: ClassExampleAST
        }
        body: {
          pos: [3, 23]
          decls: {
            pos: [4, 5]
            fields: {
              pos: [5, 9]
                [{
                  pos: [5, 9]
                  type: {
                    pos: [5, 9]
                    name: int
                    numBrackets: 0
                  }
                  varsDecls: {
                    pos: [5, 16]
                      [{
                        pos: [5, 13]
                        varDeclId: {
                          pos: [5, 13]
                          id: {
                            pos: [5, 13]
```

```
                idName: a
              }
              numBrackets: 0
            }
          } {
            pos: [5, 16]
            varDeclId: {
              pos: [5, 16]
              id: {
                pos: [5, 16]
                idName: b
              }
              numBrackets: 0
            }
            varInit: {
              pos: [5, 20]
              expr: {
                pos: [5, 20]
                value: 2
              }
            }
          }]
      }
    }]
  }
}
methods: {
  pos: [8, 5]
    [{
      pos: [8, 5]
      return :{
        pos: [8, 12]
        type: {
          pos: [8, 12]
          name: string
          numBrackets: 0
        }
      }
      methodName: {
        pos: [8, 19]
        idName: sum
      }
      params: {
        pos: [8, 23]
          [{
            pos: [8, 23]
            val: 0
            type: {
              pos: [8, 23]
              name: int
              numBrackets: 0
            }
            ids: {
              pos: [8, 27]
                [{
                  pos: [8, 27]
                  idName: c
                }]
            }
```

```
              }]
          }
          block: {
            pos: [8, 28]
            stmts: {
              pos: [11, 9]
                [{
                  pos: [9, 9]
                  what: readStmt
                  id: {
                    pos: [9, 14]
                    idName: a
                  }
                } {
                  pos: [10, 9]
                  what: readStmt
                  id: {
                    pos: [10, 14]
                    idName: b
                  }
                } {
                  pos: [11, 9]
                  what: printStmt
                  expr: {
                    pos: [11, 21]
                    op: +
                      lhs: {
                        pos: [11, 17]
                        op: +
                          lhs: {
                            pos: [11, 15]
                            id: {
                              pos: [11, 15]
                              idName: a
                            }
                          }
                        rhs: {
                          pos: [11, 19]
                          id: {
                            pos: [11, 19]
                            idName: b
                          }
                        }
                      }
                    rhs: {
                      pos: [11, 23]
                      id: {
                        pos: [11, 23]
                        idName: c
                      }
                    }
                  }
                }]
            }
          }
        }]
    }
  }
}]
```

```
  }
}
```

The ASTs for the examples in Section 2 can be found in the folder `examples/ast` in the project root.

A Appendices

A.1 Mini-Java tokens

Lexeme	Token
[".*"]	TOK_STRINGCONSTANT
;	TOK_SEMICOLON
program	TOK_PROGRAM
class	TOK_CLASS
{	TOK_LCURLY
}	TOK_RCURLY
declarations	TOK_DECLARATIONS
enddeclarations	TOK_ENDDECLARATIONS
,	TOK_COMMA
=	TOK_EQUALS
[	TOK_LSQUARE
]	TOK_RSQUARE
[]	TOK_LRSQUARE
int	TOK_INT
string	TOK_STRING
method	TOK_METHOD
void	TOK_VOID
(	TOK_LPAREN
)	TOK_RPAREN
val	TOK_VAL
.	TOK_DOT
:=	TOK_ASSIGN
return	TOK_RETURN
if	TOK_IF
else	TOK_ELSE
while	TOK_WHILE
for	TOK_FOR
switch	TOK_SWITCH
case	TOK_CASE
default	TOK_DEFAULT
print	TOK_PRINT
read	TOK_READ
<	TOK_LESS
<=	TOK_LESSEQ
==	TOK_EQEQ
!=	TOK_DIFF
>	TOK_GREATER
>=	TOK_GREATEREQ
+	TOK_PLUS
-	TOK_MINUS
||	TOK_2PIPE
*	TOK_ASTERISK
/	TOK_SLASH
%	TOK_MOD
&&	TOK_AND
@	TOK_ARROBA
not	TOK_NOT
to	TOK_TO
step	TOK_STEP
[a-zA-z_][a-zA-z0-9_]*	TOK_IDENTIFIER
[0-9]+	TOK_INTEGERCONSTANT

Table 9: Tokens of the Mini-Java language.

A.2 Quick-sort lexical analysis

Line	Column	Lexeme Size	Token	Lexeme
0	0	7	TOK_PROGRAM	program
0	8	16	TOK_IDENTIFIER	QuickSortProgram
0	24	1	TOK_SEMICOLON	;
2	0	5	TOK_CLASS	class
2	6	9	TOK_IDENTIFIER	QuickSort
3	0	1	TOK_LCURLY	{
4	1	6	TOK_METHOD	method
4	8	4	TOK_VOID	void
4	13	9	TOK_IDENTIFIER	quickSort
4	22	1	TOK_LPAREN	(
4	23	3	TOK_INT	int
4	26	1	TOK_LSQUARE	[
4	27	1	TOK_RSQUARE	]
4	29	1	TOK_IDENTIFIER	v
4	30	1	TOK_SEMICOLON	;
4	32	3	TOK_INT	int
4	36	5	TOK_IDENTIFIER	begin
4	41	1	TOK_SEMICOLON	;
4	43	3	TOK_INT	int
4	47	3	TOK_IDENTIFIER	end
4	50	1	TOK_RPAREN	)
5	2	12	TOK_DECLARATIONS	declarations
6	12	3	TOK_INT	int
6	16	1	TOK_IDENTIFIER	i
6	18	1	TOK_EQUALS	=
6	20	5	TOK_IDENTIFIER	begin
6	25	1	TOK_COMMA	,
6	27	1	TOK_IDENTIFIER	j
6	29	1	TOK_EQUALS	=
6	31	3	TOK_IDENTIFIER	end
6	34	1	TOK_MINUS	-
6	35	1	TOK_INTEGERCONSTANT	1
6	36	1	TOK_COMMA	,
6	38	5	TOK_IDENTIFIER	pivot
6	44	1	TOK_EQUALS	=
6	46	1	TOK_IDENTIFIER	v
6	47	1	TOK_LSQUARE	[
6	48	1	TOK_LPAREN	(
6	49	5	TOK_IDENTIFIER	begin
6	55	1	TOK_PLUS	+
6	57	3	TOK_IDENTIFIER	end
6	60	1	TOK_RPAREN	)
6	61	1	TOK_SLASH	/
6	62	1	TOK_INTEGERCONSTANT	2
6	63	1	TOK_RSQUARE	]
6	64	1	TOK_COMMA	,
6	66	3	TOK_IDENTIFIER	aux
6	69	1	TOK_SEMICOLON	;
7	2	15	TOK_ENDDECLARATIONS	enddeclarations
8	1	1	TOK_LCURLY	{
9	8	5	TOK_WHILE	while
9	14	1	TOK_IDENTIFIER	i
9	16	2	TOK_LESSEQ	<=
9	19	1	TOK_IDENTIFIER	j
10	8	1	TOK_LCURLY	{

Line	Column	Lexeme Size	Token	Lexeme
11	12	5	TOK_WHILE	while
11	18	1	TOK_LPAREN	(
11	19	1	TOK_IDENTIFIER	v
11	20	1	TOK_LSQUARE	[
11	21	1	TOK_IDENTIFIER	i
11	22	1	TOK_RSQUARE	]
11	24	1	TOK_LESS	<
11	26	4	TOK_IDENTIFIER	pivo
11	31	2	TOK_AND	&&
11	34	1	TOK_IDENTIFIER	i
11	36	1	TOK_LESS	<
11	38	3	TOK_IDENTIFIER	end
11	41	1	TOK_RPAREN	)
12	12	1	TOK_LCURLY	{
13	16	1	TOK_IDENTIFIER	i
13	18	2	TOK_ASSIGN	:=
13	21	1	TOK_IDENTIFIER	i
13	23	1	TOK_PLUS	+
13	25	1	TOK_INTEGERCONSTANT	1
14	12	1	TOK_RCURLY	}
14	13	1	TOK_SEMICOLON	;
15	12	5	TOK_WHILE	while
15	18	1	TOK_LPAREN	(
15	19	1	TOK_IDENTIFIER	v
15	20	1	TOK_LSQUARE	[
15	21	1	TOK_IDENTIFIER	j
15	22	1	TOK_RSQUARE	]
15	24	1	TOK_GREATER	>
15	26	4	TOK_IDENTIFIER	pivo
15	31	2	TOK_AND	&&
15	34	1	TOK_IDENTIFIER	j
15	36	1	TOK_GREATER	>
15	38	5	TOK_IDENTIFIER	begin
15	43	1	TOK_RPAREN	)
16	12	1	TOK_LCURLY	{
17	16	1	TOK_IDENTIFIER	j
17	18	2	TOK_ASSIGN	:=
17	21	1	TOK_IDENTIFIER	j
17	23	1	TOK_MINUS	-
17	25	1	TOK_INTEGERCONSTANT	1
18	12	1	TOK_RCURLY	}
18	13	1	TOK_SEMICOLON	;
19	12	2	TOK_IF	if
19	15	1	TOK_IDENTIFIER	i
19	17	2	TOK_LESSEQ	<=
19	20	1	TOK_IDENTIFIER	j
20	12	1	TOK_LCURLY	{
21	16	3	TOK_IDENTIFIER	aux
21	20	2	TOK_ASSIGN	:=
21	23	1	TOK_IDENTIFIER	v
21	24	1	TOK_LSQUARE	[
21	25	1	TOK_IDENTIFIER	i
21	26	1	TOK_RSQUARE	]
21	27	1	TOK_SEMICOLON	;
22	16	1	TOK_IDENTIFIER	v
22	17	1	TOK_LSQUARE	[
22	18	1	TOK_IDENTIFIER	i
22	19	1	TOK_RSQUARE	]
22	21	2	TOK_ASSIGN	:=

Line	Column	Lexeme Size	Token	Lexeme
22	24	1	TOK_IDENTIFIER	v
22	25	1	TOK_LSQUARE	[
22	26	1	TOK_IDENTIFIER	j
22	27	1	TOK_RSQUARE	]
22	28	1	TOK_SEMICOLON	;
23	16	1	TOK_IDENTIFIER	v
23	17	1	TOK_LSQUARE	[
23	18	1	TOK_IDENTIFIER	j
23	19	1	TOK_RSQUARE	]
23	21	2	TOK_ASSIGN	:=
23	24	3	TOK_IDENTIFIER	aux
23	27	1	TOK_SEMICOLON	;
24	16	1	TOK_IDENTIFIER	i
24	18	2	TOK_ASSIGN	:=
24	21	1	TOK_IDENTIFIER	i
24	23	1	TOK_PLUS	+
24	25	1	TOK_INTEGERCONSTANT	1
24	26	1	TOK_SEMICOLON	;
25	16	1	TOK_IDENTIFIER	j
25	18	1	TOK_EQUALS	=
25	20	1	TOK_IDENTIFIER	j
25	22	1	TOK_MINUS	-
25	24	1	TOK_INTEGERCONSTANT	1
26	12	1	TOK_RCURLY	}
27	8	1	TOK_RCURLY	}
27	9	1	TOK_SEMICOLON	;
28	8	2	TOK_IF	if
28	11	1	TOK_IDENTIFIER	j
28	13	1	TOK_GREATER	>
28	15	5	TOK_IDENTIFIER	begin
29	8	1	TOK_LCURLY	{
30	12	9	TOK_IDENTIFIER	quickSort
30	21	1	TOK_LPAREN	(
30	22	1	TOK_IDENTIFIER	v
30	23	1	TOK_COMMA	,
30	25	5	TOK_IDENTIFIER	begin
30	30	1	TOK_COMMA	,
30	32	1	TOK_IDENTIFIER	j
30	34	1	TOK_PLUS	+
30	36	1	TOK_INTEGERCONSTANT	1
30	37	1	TOK_RPAREN	)
31	8	1	TOK_RCURLY	}
31	9	1	TOK_SEMICOLON	;
32	8	2	TOK_IF	if
32	11	1	TOK_IDENTIFIER	i
32	13	1	TOK_LESS	<
32	15	3	TOK_IDENTIFIER	end
33	8	1	TOK_LCURLY	{
34	12	9	TOK_IDENTIFIER	quickSort
34	21	1	TOK_LPAREN	(
34	22	1	TOK_IDENTIFIER	v
34	23	1	TOK_COMMA	,
34	25	1	TOK_IDENTIFIER	i
34	26	1	TOK_COMMA	,
34	28	3	TOK_IDENTIFIER	end
34	31	1	TOK_RPAREN	)
35	8	1	TOK_RCURLY	}
36	4	1	TOK_RCURLY	}
37	0	1	TOK_RCURLY	}

A.3 First Set

Non Terminal	First Set
PROGRAM	program
CLASS-DECL-LIST	λ,class
CLASS-DECL	class
CLASS-BODY	{
DECLS-OPT	λ,declarations
DECLS	declarations
METHOD-DECL-LIST	λ,method
FIELD-DECL-LIST-DECLS	λ,id,int,string
FIELD-DECL	id,int,string
FIELD-DECL-AUX1	=, , ,λ
FIELD-DECL-AUX2	, ,λ
TYPE	id,int,string
TYPE-AUX	id,int,string
BRACKETS-OPT	[],λ
METHOD-DECL	method
METHOD-RETURN-TYPE	void,id,int,string
FORMAL-PARAMS-LIST	val,id,int,string
FORMAL-PARAMS-LIST-AUX	;,λ
ID-LIST-COMMA	, ,λ
FORMAL-PARAMS-LIST-OPT	λ,val,id,int,string
VAR-DECL-ID	id
VAR-INIT	{,@,(,+,-,not,num,str,id
ARRAY-INIT	{
VAR-INIT-LIST-COMMA	, ,λ
ARRAY-CREATION-EXPR	@
ARRAY-DIM-DECL	[
ARRAY-DIM-DECL-LIST	λ,[
BLOCK	declarations,{
STMT-LIST	{
STMT-LIST-SEMICOLON	;,λ
STMT	id,return,if,while,for,switch,print,read
VARIABLE-START-STMT	:=,(
ASSIGN-STMT	:=
METHOD-CALL-STMT	(
ACTUAL-PARAMS-LIST	λ,(,+,-,not,num,str,id
EXPRESSION-LIST-COMMA	, ,λ
RETURN-STMT	return
EXPRESSION-OPT	λ,(,+,-,not,num,str,id
IF-STMT	if
ELSE-PART	else,λ
IF-STMT-AUX	if,{
FOR-STMT	for
FOR-INIT-EXPR	id
STEP-OPT	step,λ
WHILE-STMT	while
SWITCH-STMT	switch
CASE	case
CASE-LIST	default,λ,case
PRINT-STMT	print
READ-STMT	read
EXPRESSION	(,+,-,not,num,str,id
REL-EXPR	(,+,-,not,num,str,id
REL-EXPR-AUX	λ,<,<=,==,!=,>=,>
ADD-EXPR	(,+,-,not,num,str,id
ADD-EXPR-AUX	λ,+,-,||

<table>
<tr><th>Non Terminal</th><th>First Set</th></tr>
<tr><td>MULT-EXPR</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>MULT-EXPR-AUX</td><td>λ,*,/,&&,%</td></tr>
<tr><td>UNARY-EXPR</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>METHOD-CALL-OPT</td><td>λ,(</td></tr>
<tr><td>REL-OP</td><td><,<=,==,!=,>=,></td></tr>
<tr><td>UNARY-OP</td><td>+,-,not</td></tr>
<tr><td>ADD-OP</td><td>+,-,||</td></tr>
<tr><td>MULT-OP</td><td>*,/,&&,%</td></tr>
<tr><td>UNSIG-LIT</td><td>num,str</td></tr>
<tr><td>VARIABLE</td><td>id</td></tr>
<tr><td>VARIABLE-AUX</td><td>.,λ,[</td></tr>
</table>

A.4 Follow Set

<table>
<tr><th>Non Terminal</th><th>Follow Set</th></tr>
<tr><td>PROGRAM</td><td>$</td></tr>
<tr><td>CLASS-DECL-LIST</td><td>$</td></tr>
<tr><td>CLASS-DECL</td><td>class,$</td></tr>
<tr><td>CLASS-BODY</td><td>class,$</td></tr>
<tr><td>DECLS-OPT</td><td>method,},{</td></tr>
<tr><td>DECLS</td><td>method,},{</td></tr>
<tr><td>METHOD-DECL-LIST</td><td>}</td></tr>
<tr><td>FIELD-DECL-LIST-DECLS</td><td>enddeclarations</td></tr>
<tr><td>FIELD-DECL</td><td>;</td></tr>
<tr><td>FIELD-DECL-AUX1</td><td>;</td></tr>
<tr><td>FIELD-DECL-AUX2</td><td>;</td></tr>
<tr><td>TYPE</td><td>id,[</td></tr>
<tr><td>TYPE-AUX</td><td>[],id,[</td></tr>
<tr><td>BRACKETS-OPT</td><td>id,=, , ,;,[</td></tr>
<tr><td>METHOD-DECL</td><td>method,}</td></tr>
<tr><td>METHOD-RETURN-TYPE</td><td>id</td></tr>
<tr><td>FORMAL-PARAMS-LIST</td><td>)</td></tr>
<tr><td>FORMAL-PARAMS-LIST-AUX</td><td>)</td></tr>
<tr><td>ID-LIST-COMMA</td><td>;,)</td></tr>
<tr><td>FORMAL-PARAMS-LIST-OPT</td><td>)</td></tr>
<tr><td>VAR-DECL-ID</td><td>=, , ,;</td></tr>
<tr><td>VAR-INIT</td><td>,,;,}</td></tr>
<tr><td>ARRAY-INIT</td><td>,,;,}</td></tr>
<tr><td>VAR-INIT-LIST-COMMA</td><td>}</td></tr>
<tr><td>ARRAY-CREATION-EXPR</td><td>,,;,}</td></tr>
<tr><td>ARRAY-DIM-DECL</td><td>[, , ,;,}</td></tr>
<tr><td>ARRAY-DIM-DECL-LIST</td><td>,,;,}</td></tr>
<tr><td>BLOCK</td><td>method,}</td></tr>
<tr><td>STMT-LIST</td><td>method,},else,;,default,case</td></tr>
<tr><td>STMT-LIST-SEMICOLON</td><td>}</td></tr>
<tr><td>STMT</td><td>;,}</td></tr>
<tr><td>VARIABLE-START-STMT</td><td>;,}</td></tr>
<tr><td>ASSIGN-STMT</td><td>;,},to</td></tr>
<tr><td>METHOD-CALL-STMT</td><td>;,},*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,],),{,step,to</td></tr>
<tr><td>ACTUAL-PARAMS-LIST</td><td>)</td></tr>
<tr><td>EXPRESSION-LIST-COMMA</td><td>),]</td></tr>
<tr><td>RETURN-STMT</td><td>;,}</td></tr>
<tr><td>EXPRESSION-OPT</td><td>;,}</td></tr>
<tr><td>IF-STMT</td><td>;,}</td></tr>
<tr><td>ELSE-PART</td><td>;,}</td></tr>
<tr><td>IF-STMT-AUX</td><td>;,}</td></tr>
</table>

<table>
<tr><th>Non Terminal</th><th>Follow Set</th></tr>
<tr><td>FOR-STMT</td><td>;,}</td></tr>
<tr><td>FOR-INIT-EXPR</td><td>to</td></tr>
<tr><td>STEP-OPT</td><td>{</td></tr>
<tr><td>WHILE-STMT</td><td>;,}</td></tr>
<tr><td>SWITCH-STMT</td><td>;,}</td></tr>
<tr><td>CASE</td><td>default,case,}</td></tr>
<tr><td>CASE-LIST</td><td>}</td></tr>
<tr><td>PRINT-STMT</td><td>;,}</td></tr>
<tr><td>READ-STMT</td><td>;,}</td></tr>
<tr><td>EXPRESSION</td><td>,,;,],},),{,step,to</td></tr>
<tr><td>REL-EXPR</td><td>,,;,],},),{,step,to</td></tr>
<tr><td>REL-EXPR-AUX</td><td>,,;,],},),{,step,to</td></tr>
<tr><td>ADD-EXPR</td><td><,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>ADD-EXPR-AUX</td><td><,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>MULT-EXPR</td><td>+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>MULT-EXPR-AUX</td><td>+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>UNARY-EXPR</td><td>*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>METHOD-CALL-OPT</td><td>*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>REL-OP</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>UNARY-OP</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>ADD-OP</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>MULT-OP</td><td>(,+,-,not,num,str,id</td></tr>
<tr><td>UNSIG-LIT</td><td>*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>VARIABLE</td><td>:=,(,*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
<tr><td>VARIABLE-AUX</td><td>:=,(,*,/,&&,%,+,-,||,<,<=,==,!=,>=,>, , ,;,],},),{,step,to</td></tr>
</table>

A.5 Numbered Grammar

Number	Grammar rule
0	PROGRAM $\models$ program id ; CLASS-DECL CLASS-DECL-LIST
1	CLASS-DECL-LIST $\models$ CLASS-DECL CLASS-DECL-LIST
2	CLASS-DECL-LIST $\models$ λ
3	CLASS-DECL $\models$ class id CLASS-BODY
4	CLASS-BODY $\models$ { DECLS-OPT METHOD-DECL-LIST }
5	DECLS-OPT $\models$ DECLS
6	DECLS-OPT $\models$ λ
7	DECLS $\models$ declarations FIELD-DECL-LIST-DECLS enddeclarations
8	METHOD-DECL-LIST $\models$ METHOD-DECL METHOD-DECL-LIST
9	METHOD-DECL-LIST $\models$ λ
10	FIELD-DECL-LIST-DECLS $\models$ FIELD-DECL ; FIELD-DECL-LIST-DECLS
11	FIELD-DECL-LIST-DECLS $\models$ λ
12	FIELD-DECL $\models$ TYPE VAR-DECL-ID FIELD-DECL-AUX1
13	FIELD-DECL-AUX1 $\models$ FIELD-DECL-AUX2
14	FIELD-DECL-AUX1 $\models$ = VAR-INIT FIELD-DECL-AUX2
15	FIELD-DECL-AUX2 $\models$, VAR-DECL-ID FIELD-DECL-AUX1
16	FIELD-DECL-AUX2 $\models$ λ
17	TYPE $\models$ TYPE-AUX BRACKETS-OPT
18	TYPE-AUX $\models$ id
19	TYPE-AUX $\models$ int
20	TYPE-AUX $\models$ string
21	BRACKETS-OPT $\models$ [] BRACKETS-OPT
22	BRACKETS-OPT $\models$ λ
23	METHOD-DECL $\models$ method METHOD-RETURN-TYPE id (FORMAL-PARAMS-LIST-OPT) BLOCK
24	METHOD-RETURN-TYPE $\models$ void
25	METHOD-RETURN-TYPE $\models$ TYPE
26	FORMAL-PARAMS-LIST $\models$ val TYPE id ID-LIST-COMMA FORMAL-PARAMS-LIST-AUX
27	FORMAL-PARAMS-LIST $\models$ TYPE id ID-LIST-COMMA FORMAL-PARAMS-LIST-AUX
28	FORMAL-PARAMS-LIST-AUX $\models$; FORMAL-PARAMS-LIST
29	FORMAL-PARAMS-LIST-AUX $\models$ λ
30	ID-LIST-COMMA $\models$, id ID-LIST-COMMA
31	ID-LIST-COMMA $\models$ λ
32	FORMAL-PARAMS-LIST-OPT $\models$ FORMAL-PARAMS-LIST
33	FORMAL-PARAMS-LIST-OPT $\models$ λ
34	VAR-DECL-ID $\models$ id BRACKETS-OPT
35	VAR-INIT $\models$ EXPRESSION
36	VAR-INIT $\models$ ARRAY-INIT
37	VAR-INIT $\models$ ARRAY-CREATION-EXPR
38	ARRAY-INIT $\models$ { VAR-INIT VAR-INIT-LIST-COMMA }
39	VAR-INIT-LIST-COMMA $\models$, VAR-INIT VAR-INIT-LIST-COMMA
40	VAR-INIT-LIST-COMMA $\models$ λ
41	ARRAY-CREATION-EXPR $\models$ @ TYPE ARRAY-DIM-DECL ARRAY-DIM-DECL-LIST
42	ARRAY-DIM-DECL $\models$ [EXPRESSION]
43	ARRAY-DIM-DECL-LIST $\models$ ARRAY-DIM-DECL ARRAY-DIM-DECL-LIST
44	ARRAY-DIM-DECL-LIST $\models$ λ
45	BLOCK $\models$ DECLS-OPT STMT-LIST
46	STMT-LIST $\models$ { STMT STMT-LIST-SEMICOLON }
47	STMT-LIST-SEMICOLON $\models$; STMT STMT-LIST-SEMICOLON
48	STMT-LIST-SEMICOLON $\models$ λ
49	STMT $\models$ VARIABLE VARIABLE-START-STMT
50	STMT $\models$ RETURN-STMT

Number	Grammar rule
51	STMT $\models$ IF-STMT
52	STMT $\models$ WHILE-STMT
53	STMT $\models$ FOR-STMT
54	STMT $\models$ SWITCH-STMT
55	STMT $\models$ PRINT-STMT
56	STMT $\models$ READ-STMT
57	VARIABLE-START-STMT $\models$ ASSIGN-STMT
58	VARIABLE-START-STMT $\models$ METHOD-CALL-STMT
59	ASSIGN-STMT $\models$:= EXPRESSION
60	METHOD-CALL-STMT $\models$ (ACTUAL-PARAMS-LIST)
61	ACTUAL-PARAMS-LIST $\models$ EXPRESSION EXPRESSION-LIST-COMMA
62	ACTUAL-PARAMS-LIST $\models$ λ
63	EXPRESSION-LIST-COMMA $\models$, EXPRESSION EXPRESSION-LIST-COMMA
64	EXPRESSION-LIST-COMMA $\models$ λ
65	RETURN-STMT $\models$ return EXPRESSION-OPT
66	EXPRESSION-OPT $\models$ EXPRESSION
67	EXPRESSION-OPT $\models$ λ
68	IF-STMT $\models$ if EXPRESSION STMT-LIST ELSE-PART
69	ELSE-PART $\models$ else IF-STMT-AUX
70	ELSE-PART $\models$ λ
71	IF-STMT-AUX $\models$ IF-STMT
72	IF-STMT-AUX $\models$ STMT-LIST
73	FOR-STMT $\models$ for FOR-INIT-EXPR to EXPRESSION STEP-OPT STMT-LIST
74	FOR-INIT-EXPR $\models$ id ASSIGN-STMT
75	STEP-OPT $\models$ step EXPRESSION
76	STEP-OPT $\models$ λ
77	WHILE-STMT $\models$ while EXPRESSION STMT-LIST
78	SWITCH-STMT $\models$ switch EXPRESSION { CASE CASE-LIST }
79	CASE $\models$ case EXPRESSION STMT-LIST
80	CASE-LIST $\models$ CASE CASE-LIST
81	CASE-LIST $\models$ default STMT-LIST
82	CASE-LIST $\models$ λ
83	PRINT-STMT $\models$ print EXPRESSION
84	READ-STMT $\models$ read id
85	EXPRESSION $\models$ REL-EXPR
86	REL-EXPR $\models$ ADD-EXPR REL-EXPR-AUX
87	REL-EXPR-AUX $\models$ REL-OP ADD-EXPR
88	REL-EXPR-AUX $\models$ λ
89	ADD-EXPR $\models$ MULT-EXPR ADD-EXPR-AUX
90	ADD-EXPR-AUX $\models$ ADD-OP MULT-EXPR ADD-EXPR-AUX
91	ADD-EXPR-AUX $\models$ λ
92	MULT-EXPR $\models$ UNARY-EXPR MULT-EXPR-AUX
93	MULT-EXPR-AUX $\models$ MULT-OP UNARY-EXPR MULT-EXPR-AUX
94	MULT-EXPR-AUX $\models$ λ

Number	Grammar rule
95	UNARY-EXPR $\models$ UNARY-OP UNARY-EXPR
96	UNARY-EXPR $\models$ UNSIG-LIT
97	UNARY-EXPR $\models$ VARIABLE METHOD-CALL-OPT
98	UNARY-EXPR $\models$ (EXPRESSION)
99	METHOD-CALL-OPT $\models$ METHOD-CALL-STMT
100	METHOD-CALL-OPT $\models$ λ
101	REL-OP $\models$ $<$
102	REL-OP $\models$ $<=$
103	REL-OP $\models$ $==$
104	REL-OP $\models$ $!=$
105	REL-OP $\models$ $>=$
106	REL-OP $\models$ $>$
107	UNARY-OP $\models$ $+$
108	UNARY-OP $\models$ -
109	UNARY-OP $\models$ not
110	ADD-OP $\models$ $+$
111	ADD-OP $\models$ -
112	ADD-OP $\models$ $\|\|$
113	MULT-OP $\models$ *
114	MULT-OP $\models$ $/$
115	MULT-OP $\models$ &&
116	MULT-OP $\models$ %
117	UNSIG-LIT $\models$ num
118	UNSIG-LIT $\models$ str
119	VARIABLE $\models$ id VARIABLE-AUX
120	VARIABLE-AUX $\models$. id VARIABLE-AUX
121	VARIABLE-AUX $\models$ λ
122	VARIABLE-AUX $\models$ [EXPRESSION EXPRESSION-LIST-COMMA] VARIABLE-AUX

A.6 Parse Table

Non-terminal	Grammar rules
PROGRAM	(program, 0)
CLASS_DECL_LIST	(class, 1),($, 2)
CLASS_DECL	(class, 3)
CLASS_BODY	({, 4)
DECLS_OPT	({, 6),(}, 6),(declarations, 5),(method, 6)
DECLS	(declarations, 7)
METHOD_DECL_LIST	(}, 9),(method, 8)
FIELD_DECL_LIST_DECLS	(id, 10),(enddeclarations, 11),(int, 10),(string, 10)
FIELD_DECL	(id, 12),(int, 12),(string, 12)
FIELD_DECL_AUX1	(;, 13),(=, 14),(„ 13)
FIELD_DECL_AUX2	(;, 16),(„ 15)
TYPE	(id, 17),(int, 17),(string, 17)
TYPE_AUX	(id, 18),(int, 19),(string, 20)
BRACKETS_OPT	(id, 22),(;, 22),(=, 22),(„ 22),([], 21),([, 22)
METHOD_DECL	(method, 23)
METHOD_RETURN_TYPE	(id, 25),(int, 25),(string, 25),(void, 24)
FORMAL_PARAMS_LIST	(id, 27),(int, 27),(string, 27),(val, 26)
FORMAL_PARAMS_LIST_AUX	(;, 28),( ), 29)
ID_LIST_,	(;, 31),(„ 30),( ), 31)
FORMAL_PARAMS_LIST_OPT	(id, 32),(int, 32),(string, 32),( ), 33),(val, 32)
VAR_DECL_ID	(id, 34)
VAR_INIT	(id, 35),({, 36),((, 35),(@, 37),(+, 35),(-, 35),(not, 35),(num, 35),(str, 35)
ARRAY_INIT	({, 38)
VAR_INIT_LIST_,	(}, 40),(„ 39)
ARRAY_CREATION_EXPR	(@, 41)
ARRAY_DIM_DECL	([, 42)
ARRAY_DIM_DECL_LIST	(;, 44),(}, 44),(„ 44),([, 43)
BLOCK	({, 45),(declarations, 45)
STMT_LIST	({, 46)
STMT_LIST_SEMICOLON	(;, 47),(}, 48)
STMT	(id, 49),(return, 50),(if, 51),(for, 53),(while, 52),(switch, 54),(print, 55),(read, 56)
VARIABLE_START_STMT	((, 58),(:=, 57)
ASSIGN_STMT	(:=, 59)
METHOD_CALL_STMT	((, 60)
ACTUAL_PARAMS_LIST	(id, 61),((, 61),( ), 62),(+, 61),(-, 61),(not, 61),(num, 61),(str, 61)
EXPRESSION_LIST_,	(„ 63),( ), 64),( ], 64)
RETURN_STMT	(return, 65)
EXPRESSION_OPT	(id, 66),(;, 67),(}, 67),((, 66),(+, 66),(-, 66),(not, 66),(num, 66),(str, 66)
IF_STMT	(if, 68)
ELSE_PART	(;, 70),(}, 70),(else, 69)
IF_STMT_AUX	({, 72),(if, 71)
FOR_STMT	(for, 73)
FOR_INIT_EXPR	(id, 74)
STEP_OPT	({, 76),(step, 75)
WHILE_STMT	(while, 77)
SWITCH_STMT	(switch, 78)
CASE	(case, 79)
CASE_LIST	(}, 82),(case, 80),(default, 81)

<table>
<tr><th>Non-Terminal</th><th>Grammar rules</th></tr>
<tr><td>PRINT_STMT</td><td>(print, 83)</td></tr>
<tr><td>READ_STMT</td><td>(read, 84)</td></tr>
<tr><td>EXPRESSION</td><td>(id, 85),((, 85),(+, 85),(-, 85),(not, 85),(num, 85),(str, 85)</td></tr>
<tr><td>REL_EXPR</td><td>(id, 86),((, 86),(+, 86),(-, 86),(not, 86),(num, 86),(str, 86)</td></tr>
<tr><td>REL_EXPR_AUX</td><td>(;, 88),({, 88),(}, 88),(,, 88),( ), 88),( ], 88),(to, 88),(step, 88),(<, 87),(<=, 87),(==, 87),(!=, 87),(>=, 87),(>, 87)</td></tr>
<tr><td>ADD_EXPR</td><td>(id, 89),((, 89),(+, 89),(-, 89),(not, 89),(num, 89),(str, 89)</td></tr>
<tr><td>ADD_EXPR_AUX</td><td>(;, 91),({, 91),(}, 91),(,, 91),( ), 91),( ], 91),(to, 91),(step, 91),(<, 91),(<=, 91),(==, 91),(!=, 91),(>=, 91),(>, 91),(+, 90),(-, 90),(||, 90)</td></tr>
<tr><td>MULT_EXPR</td><td>(id, 92),((, 92),(+, 92),(-, 92),(not, 92),(num, 92),(str, 92)</td></tr>
<tr><td>MULT_EXPR_AUX</td><td>(;, 94),({, 94),(}, 94),(,, 94),( ), 94),( ], 94),(to, 94),(step, 94),(<, 94),(<=, 94),(==, 94),(!=, 94),(>=, 94),(>, 94),(+, 94),(-, 94),(||, 94),(*, 93),(/, 93),(&&, 93),(%, 93)</td></tr>
<tr><td>UNARY_EXPR</td><td>(id, 97),((, 98),(+, 95),(-, 95),(not, 95),(num, 96),(str, 96)</td></tr>
<tr><td>METHOD_CALL_OPT</td><td>(;, 100),({, 100),(}, 100),(,, 100),((, 99),( ), 100),( ], 100),(to, 100),(step, 100),(<, 100),(<=, 100),(==, 100),(!=, 100),(>=, 100),(>, 100),(+, 100),(-, 100),(||, 100),(*, 100),(/, 100),(&&, 100),(%, 100)</td></tr>
<tr><td>REL_OP</td><td>(<, 101),(<=, 102),(==, 103),(!=, 104),(>=, 105),(>, 106)</td></tr>
<tr><td>UNARY_OP</td><td>(+, 107),(-, 108),(not, 109)</td></tr>
<tr><td>ADD_OP</td><td>(+, 110),(-, 111),(||, 112)</td></tr>
<tr><td>MULT_OP</td><td>(*, 113),(/, 114),(&&, 115),(%, 116)</td></tr>
<tr><td>UNSIG_LIT</td><td>(num, 117),(str, 118)</td></tr>
<tr><td>VARIABLE</td><td>(id, 119)</td></tr>
<tr><td>VARIABLE_AUX</td><td>(;, 121),({, 121),(}, 121),(,, 121),((, 121),( ), 121),([, 122),( ], 121),(:=, 121),(to, 121),(step, 121),(<, 121),(<=, 121),(==, 121),(!=, 121),(>=, 121),(>, 121),(+, 121),(-, 121),(||, 121),(*, 121),(/, 121),(&&, 121),(%, 121),(., 120)</td></tr>
</table>

A.7 Yacc Grammar

```
program                  : TOK_PROGRAM TOK_IDENTIFIER TOK_SEMICOLON class_decl class_decl_list
class_decl_list          : /* empty */ | class_decl class_decl_list
class_decl               : TOK_CLASS TOK_IDENTIFIER class_body
                         | TOK_CLASS error class_body
class_body               : TOK_LCURLY decls_opt method_decl_list TOK_RCURLY
                         | TOK_LCURLY error TOK_RCURLY
decls_opt                : /* empty */ | decls
decls                    : TOK_DECLARATIONS field_decl_list_decls TOK_ENDDECLARATIONS
                         | TOK_DECLARATIONS error TOK_ENDDECLARATIONS
method_decl_list         : /* empty */ | method_decl method_decl_list
field_decl_list_decls    : /* empty */ | field_decl TOK_SEMICOLON field_decl_list_decls
field_decl               : type var_decl_id field_decl_aux1
field_decl_aux1          : field_decl_aux2 | TOK_EQUALS var_init field_decl_aux2
field_decl_aux2          : /* empty */ | TOK_COMMA var_decl_id field_decl_aux1
type                     : type_aux brackets_opt
type_aux                 : TOK_IDENTIFIER | TOK_INT | TOK_STRING
brackets_opt             : /* empty */ | TOK_LRSQUARE brackets_opt
method_decl              : TOK_METHOD method_return_type TOK_IDENTIFIER TOK_LPAREN formal_params_list_opt TOK_RPAREN block
                         | TOK_METHOD method_return_type TOK_IDENTIFIER TOK_LPAREN error TOK_RPAREN block
method_return_type       : TOK_VOID | type
formal_params_list       : TOK_VAL type TOK_IDENTIFIER id_list_comma formal_params_list_aux
                         | type TOK_IDENTIFIER id_list_comma formal_params_list_aux
formal_params_list_aux   : /* empty */ | TOK_SEMICOLON formal_params_list
id_list_comma            : /* empty */ | TOK_COMMA TOK_IDENTIFIER id_list_comma
formal_params_list_opt   : /* empty */ | formal_params_list
var_decl_id              : TOK_IDENTIFIER brackets_opt
var_init                 : expr | array_init | array_creation_expr
array_init               : TOK_LCURLY var_init var_init_list_comma TOK_RCURLY
                         | TOK_LCURLY error TOK_RCURLY
var_init_list_comma      : /* empty */ | TOK_COMMA var_init var_init_list_comma
array_creation_expr      : TOK_ARROBA type array_dim_decl array_dim_decl_list
array_dim_decl           : TOK_LSQUARE expr TOK_RSQUARE
array_dim_decl_list      : /* empty */ | array_dim_decl array_dim_decl_list
block                    : decls_opt stmt_list
stmt_list                : TOK_LCURLY stmt stmt_list_semicolon TOK_RCURLY
                         | TOK_LCURLY error TOK_RCURLY
stmt_list_semicolon      : /* empty */ | TOK_SEMICOLON stmt stmt_list_semicolon
stmt                     : var var_start_stmt | return_stmt| if_stmt
                         | while_stmt | for_stmt | switch_stmt | print_stmt
                         | read_stmt
var_start_stmt           : assign_stmt | method_call_stmt
assign_stmt              : TOK_ASSIGN expr
method_call_stmt         : TOK_LPAREN actual_params_list TOK_RPAREN
                         | TOK_LPAREN error TOK_RPAREN
actual_params_list       : /* empty */ | expr expr_list_comma
expr_list_comma          : /* empty */ | TOK_COMMA expr expr_list_comma
return_stmt              : TOK_RETURN expr_opt
expr_opt                 : /* empty */ | expr
if_stmt                  : TOK_IF expr stmt_list else_part
else_part                : /* empty */ | TOK_ELSE if_stmt_aux
if_stmt_aux              : if_stmt | stmt_list
for_stmt                 : TOK_FOR for_init_expr TOK_TO expr step_opt stmt_list
for_init_expr               : TOK_IDENTIFIER assign_stmt
step_opt                 : /* empty */ | TOK_STEP expr
while_stmt               : TOK_WHILE expr stmt_list
switch_stmt              : TOK_SWITCH expr TOK_LCURLY case case_list TOK_RCURLY
                         | TOK_SWITCH error TOK_LCURLY
                         | TOK_SWITCH expr TOK_LCURLY error TOK_RCURLY
case                     : TOK_CASE expr stmt_list
                         | TOK_CASE error stmt_list
case_list                : /* empty */ | case case_list | TOK_DEFAULT stmt_list
print_stmt               : TOK_PRINT expr
read_stmt                : TOK_READ TOK_IDENTIFIER
expr                     : al_expr TOK_EQEQ al_expr
                         | al_expr TOK_LESS al_expr
                         | al_expr TOK_LESSEQ al_expr
                         | al_expr TOK_GREATEREQ al_expr
                         | al_expr TOK_GREATER al_expr
                         | al_expr TOK_DIFF al_expr
                         | al_expr
al_expr                  : TOK_PLUS al_expr %prec TOK_UPLUS
                         | TOK_MINUS al_expr %prec TOK_UMINUS
                         | TOK_NOT al_expr
                         | al_expr TOK_PLUS al_expr
                         | al_expr TOK_MINUS al_expr
                         | al_expr TOK_2PIPE al_expr
                         | al_expr TOK_ASTERISK al_expr
                         | al_expr TOK_SLASH al_expr
                         | al_expr TOK_AND al_expr
                         | al_expr TOK_MOD al_expr
                         | TOK_LPAREN expr TOK_RPAREN
                         | TOK_LPAREN error TOK_RPAREN
                         | TOK_INTEGERCONSTANT
                         | TOK_STRINGCONSTANT
                         | var method_call_opt
method_call_opt          : /* empty */ | method_call_stmt
var                      : TOK_IDENTIFIER var_aux
var_aux                  : /* empty */ | TOK_DOT TOK_IDENTIFIER var_aux
                         | TOK_LSQUARE expr expr_list_comma TOK_RSQUARE var_aux
```