

The Mini-Java Programming Language

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

September 10, 2018

Contents

1	Mini-Java Language Reference	4
1.1	Syntax	4
1.1.1	Reserved words	4
1.1.2	Valid Identifiers	4
1.1.3	Data Types	5
1.1.4	Expressions	5
1.1.5	Variables Declaration and Initialization	6
1.1.6	Assignments	6
1.1.7	Conditional structures	7
1.1.7.1	If/Else Statements	7
1.1.7.2	Switch/Case Statements	7
1.1.8	Repetition structures	7
1.1.8.1	While Statement	8
1.1.8.2	For Statement	8
1.1.9	Subprograms declaration and call	8
1.1.10	Statement blocks	9
1.1.11	Classes	10
1.1.12	IO	10
1.1.13	Comments	11
2	Examples	12
2.1	Merge Sort	12
2.2	Binary search	13
2.3	Quick Sort	13
2.4	Specific constructs	14
2.5	Program with errors	15
3	Extended Backus-Naur Form (EBNF)	16

*jacksonrauup@hotmail.com

†gustavowl@lcc.ufrn.br

‡vitorgreati@gmail.com

§joelfelipe07@gmail.com

4	mjc: a compiler for Mini-Java	18
4.1	Compiler steps	18
4.2	Lexical Analysis or Scanning	18
4.3	Syntax Analysis or Parsing	19
4.3.1	Top-down parsing	19
4.3.1.1	FIRST Set	20
4.3.1.2	FOLLOW Set	20
4.3.1.3	LL(1) Grammar	20
4.3.1.4	Recursive Predictive Parser	21
4.3.1.5	Non-Recursive Predictive Parser	22
A	Appendices	24
A.1	Mini-Java tokens	24
A.2	Quick-sort lexical analysis	25
A.3	First Set	28
A.4	Follow Set	30
A.5	Numbered Grammar	32
A.6	Parse Table	35

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.

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.

```
1 program
2 class
3 method
4 val
5 int
6 string
7 void
8 declarations
9 enddeclarations
10 if
11 else
12 for
13 to
14 step
15 while
16 switch
17 case
18 print
19 read
20 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 terms 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 1
2 +5
3 -7
4 (17)
5 90 + 90
6 -21 + 12
7 90 * 2
8 78 / 3
9 25 % 4
10 add(2, 5)
11 add(num, 5*7)
12 assume("control", 2112)
13 doMagic(i * (2 + const) - shift, (flag * 5) + year / month)
14
15 #These expressions are not standalone.
16 #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.).

```
1 73
2 73 == 1001001
3 2112 < 5
4 isEven(number) && isPrime(number) && number == 2
5 (1 || false) && true
6 length("Rush") <= length("Caress of Steel") - 13
7
```

```

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):

```

1 color
2 names[]
3 []
4 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:

```

1 string str;
2 int a, b = 2, c, d, e = 5;
3 string[] str_arr = @string[10];
4 int[] vec = @int[size + 2];
5 int[][] identity = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
6 int[][] unknown_line = { @int[2], {73, 2112} };
7 #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:

```

1 #this shall be inside a method definition (class body)
2 declarations
3     string str;
4     int a, b, c = 8;
5     string[] str_arr = @string[10];
6 enddeclarations
7

```

```

8  str := "text";
9  a := 2;
10 b := 3;
11 c := 5;
12 str_arr[a] := "a";
13 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

```

1  #this shall be inside a method definition (class body)
2  declarations
3      string str;
4      int a = 5;
5  enddeclarations
6
7  if a > 10 {
8      str := "bigger"
9  } else if a == a {
10     str := "equal"
11 } else {
12     str := "smaller"
13 }

```

1.1.7.2 Switch/Case Statements

```

1  #this shall be inside a method definition (class body)
2  declarations
3      string[] str_arr = @string[10];
4      int control = 3;
5  enddeclarations
6
7  switch(control) {
8      case 0 { str_arr[0] := "zero" }
9      case 1 { str_arr[0] := "one" }
10     case 2 { str_arr[0] := "two" }
11     case 3 { str_arr[0] := "three" }
12     case 4 { str_arr[0] := "four" }
13     default { str_arr[0] := "minus one" }
14 }

```

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

```
1 #this shall be inside a method definition (class body)
2 declarations
3     int i = 0;
4     int max = 100;
5 enddeclarations
6
7 while i < 100 {
8     i := i + 1;
9 }
```

1.1.8.2 For Statement

```
1 #this shall be inside a method definition (class body)
2 declarations
3     int max = 100;
4     int i;
5 enddeclarations
6
7 for i := 0 to max {
8     i := i + 1;
9 }
10
11 for i := max to 10 step -1 {
12     i := i - 1;
13 }
```

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:

```
1 program MethodExamples;
2
3 class Math {
4
5     declarations
6         int i = 1;
7     enddeclarations
8
9     method void toIncrease() {
10         i := i + 1
11     }
12 }
```



```

13     method void toDecrease() {
14         i := i - 1
15     }
16
17     method void toIncreaseN(int n) {
18         int j = 0;
19         while (j < n) {
20             toIncrease();
21             j := j + 1
22         }
23     }
24
25
26     method void toDecreaseN(int n) {
27         int j = 0;
28         while (j < n) {
29             toDecrease();
30             j := j + 1
31         }
32     }
33
34     method int toIncrease() {
35         i := i + 1;
36         return i
37     }
38
39     method int toDecreaseN(int n) {
40         i := i - n;
41         return i
42     }
43 }

```

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:

```

1  declarations
2      String str;
3      int a, b , c = 8;
4  enddeclarations
5
6  {
7      str := "text";
8      a := 2;
9      b := 3;
10     c := 5;
11 }

```

Next, an example of one without the declaration section:

```

1  {

```

```

2      str := "text";
3      a := 2;
4      b := 3;
5      c := 5;
6  }

```

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:

```

1  class ClassExample1
2  {
3      declarations
4          int i = 0;
5      enddeclarations
6
7      method void method1() {}
8  }
9
10 class ClassExample2
11 {
12     method void method1() {}
13 }
14
15 class ClassExample3
16 {
17     declarations
18         int i = 0;
19     enddeclarations
20 }
21
22 class ClassExample4
23 {
24 }

```

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:

```

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

```

```
4 print "Hello " + world
5 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:

```
1 #This line is commented
2 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

```
1  program MergeSortProgram;
2  class MergeSort
3  {
4      method void merge(int [] v; int begin; int middle; int end)
5          declarations
6              int i = begin, j = middle+1, k = begin, length = end-begin+1;
7              int [] aux;
8          enddeclarations
9      {
10         while (i <= middle) && (j <= end)
11         {
12             if v[i] < v[j] {
13                 aux[k] := v[i];
14                 i := i + 1
15             }
16             else
17             {
18                 aux[k] := v[j];
19                 j := j + 1
20             };
21             k := k + 1
22         };
23
24         while i <= middle
25         {
26             aux[k] := v[i];
27             i := i + 1;
28             k := k + 1
29         };
30
31         while j <= end
32         {
33             aux[k] := v[j];
34             j := j + 1;
35             k := k + 1
36         };
37
38         for k := begin to end step 1
39         {
40             aux[k] := v[k]
41         }
42     }
43     method void mergeSort(int [] v; int begin; int end)
```

```

44     declarations
45         int middle;
46     enddeclarations
47 {
48     if begin < end
49     {
50         middle := (end + begin)/2;
51         mergeSort(v, begin, middle);
52         mergeSort(v, middle, end);
53         merge(v, begin, middle, end)
54     }
55 }
56 }

```

2.2 Binary search

```

1  program BinarySearchProgram;
2
3  class BinarySearch
4  {
5      method int binarySearch(int[] arr; int l; int r; int x)
6          declarations
7              int mid;
8          enddeclarations
9      {
10         if r >= 1
11         {
12             mid := 1 + (r - 1)/2;
13             if arr[mid] == x
14             {
15                 return mid
16             }
17             else if arr[mid] > x
18             {
19                 return binarySearch(arr, l, mid-1, x)
20             };
21             return binarySearch(arr, mid+1, r, x)
22         };
23         return -1
24     }
25 }

```

2.3 Quick Sort

```

1  program QuickSortProgram;
2
3  class QuickSort
4  {
5      method void quickSort(int[] v; int begin, end)
6          declarations
7              int i = begin, j = end-1, pivot = v[(begin + end)/2], aux;

```

```

8      enddeclarations
9      {
10         while i <= j
11         {
12             while (v[i] < pivo) && (i < end)
13             {
14                 i := i + 1
15             };
16             while (v[j] > pivo) && (j > begin)
17             {
18                 j := j - 1
19             };
20             if i <= j
21             {
22                 aux := v[i];
23                 v[i] := v[j];
24                 v[j] := aux;
25                 i := i + 1;
26                 j := j - 1
27             }
28         };
29         if j > begin
30         {
31             quickSort(v, begin, j + 1)
32         };
33         if i < end
34         {
35             quickSort(v, i, end)
36         };
37         return
38     }
39 }

```

2.4 Specific constructs

```

1  program Example;
2
3  class Useless { }
4
5  class QuasiUseless {
6      declarations
7      enddeclarations
8  }
9
10 class HalfUseless {
11     declarations
12     MyClass[] v = @MyClass[10];
13     string a, b;
14     int[][] identity = { {1, 0, 0}, {0, 1, 0}, {0, 0, 1} };
15     enddeclarations
16 }

```

```

17
18 class Switch {
19     method string map(val int key) {
20         switch key {
21             case 1 {
22                 return "A"
23             }
24             case 2 {
25                 return "B"
26             }
27             default {
28                 return "Invalid"
29             }
30         };
31         print "Finished."
32     }
33 }
34
35 class Expr {
36     declarations
37     MyClass [][] v = @MyClass[10][10];
38     int a, b;
39     int bool;
40     enddeclarations
41     method void makeExpr() {
42         read a;
43         bool := not ((a != b) && b || (a || b));
44         bool := a + b - a * b / a % b + a + v[2,3].name[19];
45         a := -2;
46         a := +2;
47         makeExpr()
48     }
49 }

```

2.5 Program with errors

```

1 program string ;
2
3 class $oi {
4     declarations
5     int a = /2;
6     enddeclarations
7 }

```

3 Extended Backus-Naur Form (EBNF)

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

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

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/mjclex` 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/mjcllexer [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:

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

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:

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

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)):

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

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:

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

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/mjc111` doesn't exist, execute, from the root directory, `make 111parser`.
2. Having a program called `program.mj`, execute the following command from the root:
`./bin/mjc111 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 Appendix 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:

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

This grammar will generate the parse table found in Table 3:

	a	b	\$
S	A	bB	A
A	aA		λ
B		bB	λ

Table 3: Example of Parse Table

So, the behaviour for generating the empty string can be found in Table 4

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

Table 4: 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 5

Production Stack	Token Stack	Production Matched
S\$	bbb\$	$S \rightarrow bB$
bB\$	bbb\$	match and pop
B\$	bb\$	$B \rightarrow bB$
bB\$	bb\$	match and pop
B \$	b\$	$B \rightarrow bB$
bB\$	b\$	match and pop
B \$	\$	$B \rightarrow \lambda$
\$	\$	SUCCESS

Table 5: 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 6.

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

Table 6: 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/mjc111` doesn't exist, execute, from the root directory, `make 111parser`.
2. Having a program called `program.mj`, execute the following command from the root:
`./bin/mjc111 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 Appendix 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.
```

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 7: 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_LESEQ	<=
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	$\lambda, \text{declarations}$
DECLS	declarations
METHOD-DECL-LIST	λ, method
FIELD-DECL-LIST-DECLS	$\lambda, \text{id}, \text{int}, \text{string}$
FIELD-DECL	$\text{id}, \text{int}, \text{string}$
FIELD-DECL-AUX1	$=, , , \lambda$
FIELD-DECL-AUX2	$, , \lambda$
TYPE	$\text{id}, \text{int}, \text{string}$
TYPE-AUX	$\text{id}, \text{int}, \text{string}$
BRACKETS-OPT	$, \lambda$
METHOD-DECL	method
METHOD-RETURN-TYPE	$\text{void}, \text{id}, \text{int}, \text{string}$
FORMAL-PARAMS-LIST	$\text{val}, \text{id}, \text{int}, \text{string}$
FORMAL-PARAMS-LIST-AUX	$;; \lambda$
ID-LIST-COMMA	$, , \lambda$
FORMAL-PARAMS-LIST-OPT	$\lambda, \text{val}, \text{id}, \text{int}, \text{string}$
VAR-DECL-ID	id
VAR-INIT	$\{, @, (, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
ARRAY-INIT	{
VAR-INIT-LIST-COMMA	$, , \lambda$
ARRAY-CREATION-EXPR	@
ARRAY-DIM-DECL	[
ARRAY-DIM-DECL-LIST	$\lambda, [$
BLOCK	declarations, {
STMT-LIST	{
STMT-LIST-SEMICOLON	$;; \lambda$
STMT	$\text{id}, \text{return}, \text{if}, \text{while}, \text{for}, \text{switch}, \text{print}, \text{read}$
VARIABLE-START-STMT	$:=, ($
ASSIGN-STMT	$:=$
METHOD-CALL-STMT	(
ACTUAL-PARAMS-LIST	$\lambda, (, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
EXPRESSION-LIST-COMMA	$, , \lambda$
RETURN-STMT	return
EXPRESSION-OPT	$\lambda, (, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
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	$(, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
REL-EXPR	$(, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
REL-EXPR-AUX	$\lambda, <, <=, ==, !=, >=, >$
ADD-EXPR	$(, +, -, \text{not}, \text{num}, \text{str}, \text{id}$
ADD-EXPR-AUX	$\lambda, +, -, $

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

A.4 Follow Set

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

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 $\models \lambda$
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 $\models \lambda$
7	DECLS \models declarations FIELD-DECL-LIST-DECLS enddeclarations
8	METHOD-DECL-LIST \models METHOD-DECL METHOD-DECL-LIST
9	METHOD-DECL-LIST $\models \lambda$
10	FIELD-DECL-LIST-DECLS \models FIELD-DECL ; FIELD-DECL-LIST-DECLS
11	FIELD-DECL-LIST-DECLS $\models \lambda$
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 $\models \lambda$
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 $\models \lambda$
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 $\models \lambda$
30	ID-LIST-COMMA $\models ,$ id ID-LIST-COMMA
31	ID-LIST-COMMA $\models \lambda$
32	FORMAL-PARAMS-LIST-OPT \models FORMAL-PARAMS-LIST
33	FORMAL-PARAMS-LIST-OPT $\models \lambda$
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 $\models \lambda$
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 $\models \lambda$
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 $\models \lambda$
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)

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