The Mini-Java Programming Language

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

August 14, 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

3 Extended Backus-Naur Form (EBNF) **15**

*jacksonrauup@hotmail.com
†gustavowl@lcc.ufrn.br
‡vitorgreati@gmail.com
§joelfelipe07@gmail.com

4 mjc: a compiler for Mini-Java **17**
4.1 Front-end 17
4.1.1 Lexical Analysis or Scanning 17
4.1.2 Syntax Analysis or Parsing 18
4.2 Back-end 18

5 Appendices **19**
5.1 Mini-Java tokens 19
5.2 Quick-sort lexical analysis 20

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

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

```

```
#Relational statements are normally encapsulated in statements
#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

```

```
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 }
```

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
    }
```

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

```
44
45      method void mergeSort(int[] v; int begin; int end)
46          declarations
47              int middle;
48          enddeclarations
49      {
50          if(begin < end)
51          {
52              middle := (end + begin)/2;
53              mergeSort(v, begin, middle);
54              mergeSort(v, middle, end);
55              merge(v, begin, middle, end)
56          }
57      {
58  }
```

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 := l + (r - l)/2;
13              if (arr[mid] == x)
14              {
15                  return mid
16              };
17
18              if (arr[mid] > x)
19              {
20                  return binarySearch(arr, l, mid-1, x)
21              };
22              return binarySearch(arr, mid+1, r, x)
23          };
24          return -1
25      }
26  }
```

2.3 Quick Sort

```
1   program QuickSortProgram;
2
3   class QuickSort
4   {
```

```
5       method void quickSort(int[] v; int begin; int 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      }
38  }
```

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⟩} "}"
⟨case⟩ ⊨ "case" ⟨expression⟩ ⟨stmt-list⟩
⟨expression⟩ ⊨ ⟨simple-expr⟩ [⟨rel-op⟩ ⟨simple-expr⟩]
⟨rel-op⟩ ⊨ "<" | "<=" | "==" | "!=" | ">=" | ">"
⟨simple-expr⟩ ⊨ [⟨simple-un-op⟩] ⟨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⟩
⟨unsig-lit⟩ ⊨ ⟨integer-lit⟩ | ⟨string-lit⟩

⟨variable⟩ ⊨ ⟨id⟩ {⟨variable-aux⟩}

⟨variable-aux⟩ ⊨ "[" ⟨expression⟩ {"," ⟨expression⟩} "]" | "." ⟨id⟩

⟨print-stmt⟩ ⊨ "print" ⟨expression⟩

⟨read-stmt⟩ ⊨ "read" ⟨id⟩

4 mjc: a compiler for Mini-Java

4.1 Front-end

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*. The following subsections detail each one's peculiarities and present how they are implemented in *mjc*.

4.1.1 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 5.1.

`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 5.2. 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.

4.1.2 Syntax Analysis or Parsing

To be written in some of the next versions.

4.2 Back-end

To be written in some of the next versions.

5 Appendices

5.1 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
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
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
not	TOK_NOT
to	TOK_TO
step	TOK_STEP
[a-zA-z_][a-zA-z0-9_]*	TOK_IDENTIFIER
[0-9]+	TOK_INTEGERCONSTANT

Table 2: Tokens of the Mini-Java language.

5.2 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	}