1. Introduction

This document describes the MicroJava language. MicroJava is similar to Java but much simpler.

2. General Characteristics

A MicroJava program consists of a single program file with static fields and static methods. There are no external classes but only classes that can be used as data types. The main method of a MicroJava program is always called **main()**. When a Micro-Java program is called this method is executed. There are:

- Constants of type **int** (e.g. 3), **char** (e.g. 'x') and **bool** (e.g. true).
- Variables which can be global (static), local and fields (within a class).
- Primitive types: **int**, **char** (Ascii), **bool**
- Structural/reference types which are one-dimensional arrays like in Java and classes with fields and methods. Variables of this type represent references (they contain valid addresses which cannot be changed explicitly).
- Static methods in the main program and classes.

There is no garbage collector (allocated objects are only deallocated when the pro-gram ends). MicroJava supports inheritance and method overriding. Class methods are bound to a class instance and have an implicit parameter **this** (reference to a class instance for which the method was called). Reference variable **this** is implicitly declared inside class methods as the first formal argument. Its type is a reference to a class in which the method is declared. Predeclared procedures are **ord**, **chr**, **len**.

3. Sample program

```
program P
const int size = 10;

class Table {
        int pos[], neg[];
        {
                void putp ( int a, int idx ) { this.pos[idx] = a; }
                void putn ( int a, int idx ) { this.neg[idx] = a; }
                int getp ( int idx ) { return this.pos[idx]; }
                int getn ( int idx ) { return this.neg[idx]; }
        }
}
Table val;

void main ( )  int x, i;
{
        //---------- Initialize val
        val     = new Table;
        val.pos = new int [size];
        val.neg = new int [size];

        for ( i = 0;  i < size;  i++ ) {
                val.putn ( 0, i );
                val.putp ( 0, i );
        }

        //---------- Read values
        read ( x );
        for ( ;x > 0; ) {
                if ( 0 <= x && x < size ) {
                        val.putp ( val.getp ( x ) + 1 );
                } else if ( -size < x && x < 0 ) {
                        val.putn( val.getn ( -x ) + 1 );
                }
                read ( x );
        }
}
```

4. Syntax

```
Program              = ″program″ ident {ConstDecl | VarDecl | ClassDecl} ″{″ {MethodDecl} ″}″.
ConstDecl            = ″const″ Type ident ″=″ (numConst | charConst | boolConst) {, ident ″=″ (numConst | charConst | boolConst)} ″;″.
VarDecl              = Type ident [″[″ ″]″] {″,″ ident [″[″ ″]″]} ″;″.
ClassVarDecl         = Type ident [″[″ ″]″] {″,″ ident [″[″ ″]″]} ″;″.
ClassDecl            = ″class″ ident [″extends″ Type] ″{″ {ClassVarDecl} [″{″ {MethodDecl} ″}″] ″}″.
MethodDecl           = ["static"] (Type | ″void″) ident ″(″ [FormPars] ″)″ {VarDecl} ″{″{Statement} ″}″.
FormPars             = Type ident [″[″ ″]″] {″,″ Type ident [″[″ ″]″]}.
Type                 = ident.
Statement            = DesignatorStatement ″;″
                     | ″if″ ″(″ Condition ″)″ Statement [″else″ Statement]
                     | ″for″ ″(″ [DesignatorStatement ] ″;″ [Condition] ″;″ [DesignatorStatement] ″)″ Statement
                     | ″break″ ″;″
                     | ″continue″ ″;″
                     | ″return″ [Expr] ″;″
                     | ″read″ ″(″ Designator ″)″ ″;″
                     | ″print″ ″(″ Expr [″,″ numConst] ″)″ ″;″
                     | ″{″ {Statement} ″}″.
DesignatorStatement  = Designator (Assignop Expr | ″(″ [ActPars] ″)″ | ″++″ | ″--″)
ActPars              = Expr {″,″ Expr}.
Condition            = CondTerm {″||″ CondTerm}.
CondTerm             = CondFact {″&&″ CondFact}.
CondFact             = Expr [Relop Expr].
Expr                 = [″-″] Term {Addop Term}.
Term                 = Factor {Mulop Factor}.
Factor               = Designator [″(″ [ActPars] ″)″]
                     | numConst
                     | charConst
                     | boolConst
                     |″new″ Type [″[″ Expr ″]″]
                     | ″(″ Expr ″)″.
Designator           = ident {″.″ ident | ″[″ Expr ″]″}.
Assignop             = ″=″ | AddopRight | MulopRight.
Relop                = ″==″ | ″!=″ | ″>″ | ″>=″ | ″<″ | ″<=″.
Addop                = AddopLeft|AddopRight.
AddopLeft            = ″+″ | ″-″.
AddopRight           = ″+=″ | ″-=″.
Mulop                = MulopLeft | MulopRight .
MulopLeft            = ″*″ | ″/″ | ″%″.
MulopRight           = ″*=″| ″/=″ | ″%=″.
```

5. Lexical structure

Keywords:	program, break, class, else, const, if, new, print, read, return, void, for, extends, continue, static
Token types:	ident = letter {letter | digit | ″_″}.
	numConst = digit {digit}.
	charConst = ″′″ printableChar ″′″.
	boolConst = (″true″ | ″false″).
Operators:	+, -, *, /, %, ==, !=, >, >=, <, <=, &&, ||, =, +=, -=, *=, /=, %=, ++, --, ;, comma, ., (,), [,], {, }
Comments:	// until the end of line

6. Semantics

All terms in this document that have a definition are underlined to emphasize their special meaning. The definitions of these terms are given hereafter.

Reference type

Arrays and classes are called reference types.

Type of a constant

The type of an integer constant (e.g. **17**) is **int**.

The type of a character constant (e.g. **'x'**) is **char**.

The type of a boolean constant (e.g. **true**) is **bool**.

Equivalent data types

Two data types are equivalent if they are denoted by the same type name, or if both types are arrays and their element types are the same.

Type compatibility

Two types are compatible if they are the same, or if one of them is a reference type and the other is the type of null.

Assignment compatibility

A type src is assignment compatible with a type dst if:

- src and dsc types are equivalent
- dst type is a reference type and src type is null
- dst is a reference to a base class and src is a reference to some derived class

Predeclared identifiers

- **int**, type for all integer values
- **char**, type for all character values
- **bool**, logic type
- **null**, a value of a class or array variable, meaning "pointing to no value"
- **eol**, end of line (equivalent to '\n'), print (eol) moves to cursor to a new line
- **chr**, standard method, chr(i) converts the int expression i into a char value
- **ord**, standard method, ord(ch) converts the char value ch into an int value
- **len**, standard methiod, len(a) returns the number of elements of the array a

Scope

A scope is the textual range of a method or a class. It extends from the point after the declaring method or class name to the closing curly bracket of the method or class declaration. A scope excludes other scopes that are nested within it. We assume that there is an (artificial) outermost scope (called the **universe**), to which the **main** method is local and which contains all predeclared names. The declaration of a name in an inner scope hides the declarations of the same name in outer scopes.

Note

Indirectly recursive methods are not allowed, since every name must be declared before it is used. This would not be possible if indirect recursion were allowed. A predeclared name (e.g. **int** or **char**) can be redeclared in an inner scope (but this is not recommended).

7. Context Conditions

7.1 General context conditions

- Every name must be declared before it is used.
- A name must not be declared twice in the same scope.
- A program must contain a method named **main**. It must be declared as a void method and must not have parameters.

7.2 Context conditions for standard methods

- **chr(e)**, **e** must be an expression of type int.
- **ord(c)**, **c** must be of type char.
- **len(a)**, **a** must be an array.

7.3 Context conditions for the MicroJava productions

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

ConstDecl = ″const″ Type ident ″=″ (numConst | charConst | boolConst) ″;″.

- The type of **numConst**, **charConst** or **boolConst** must be equivalent to type **Type**.

VarDecl = Type ident [″[″ ″]″] {″,″ ident [″[″ ″]″]} ″;″.

ClassVarDecl = Type ident [″[″ ″]″] {″,″ ident [″[″ ″]″]} ″;″.

ClassDecl = ″class″ ident [″extends″ Type] ″{″ {ClassVarDecl} [″{″ {MethodDecl} ″}″] ″}″.

- Type **Type** must be an class declared in the main program.

MethodDecl = [static](Type | ″void″) ident ″(″ [FormPars] ″)″ {VarDecl} ″{″ {Statement} ″}″.

- If a method is a function it must be left via a return statement (this is checked at run time).
- Keyword **static** is used only for static methods of classes (it is not used for global functions).
- Static methods can use global variables and invoke global functions, as well as other static methods in the current scope.
- Static methods cannot be redefined in derived classes.
- Static methods and global functions do not have an implicit parameter **this**.

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

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

- **ident** must denote a type.

Statement = DesignatorStatement ″;″.

DesignatorStatement = Designator Assignop Expr ";".

- **Designator** must denote a variable, an array element or an object field.
- The type of **Expr** must be assignment compatible with the type of **Designator**.

DesignatorStatement = Designator (″++″ | ″--″) ″;″.

- **Designator** must denote a variable, an array element or an object field.
- **Designator** must be of type **int**.

DesignatorStatement = Designator ″(″ [ActPars] ″)″ ″;″.

- **Designator** must denote a static or non static method of a class or a global function of the program.
- Static method can be invoked with an object of a class or with a name of the class that it belongs to.

Statement = ″break″.

- Statement **break** can only be used within a loop. It stops the execution of a closest outer loop.

Statement = ″continue″.

- Statement **continue** can only be used within a loop. It stops the execution of the current iteration of a closest outer loop.

Statement = "read" "(" Designator ")" ";".

- **Designator** must denote a variable, an array element or an object field.
- **Designator** must be of type **int**, **char** or **bool**.

Statement = "print" "(" Expr ["," number] ")" ";".

- **Expr** must be of type **int**, **char** or **bool**.

Statement = "return" [Expr] .

- The type of **Expr** must be assignment compatible with the return type of the current method or global function.
- If **Expr** is missing the current method must be declared as **void**.
- This statement cannot be used outside of methods and global functions.

Statement = ″if″ ″(″ Condition ″)″ Statement [″else″ Statement]

- If the value of the qualifying expression **Condition** is true, the statements in the **if** branch are executed, otherwise the statements in the **else** branch are executed.
- The type of the the qualifying expression **Condition** must be **bool**.

Statement=″for″ ″(″[DesignatorStatement]″;″[Condition] ″;″ [DesignatorStatement] ″)″ Statement

- The type of the the qualifying expression **Condition** must be **bool**.
- If stated, first the statement described with the non terminal **DesignatorStatement** is executed, and only once (it is not part of the cyclic execution).
- At the beginning of each iteration the value of the qualifying expression **Condition** is checked. If it is not stated it is considered **true**. Value **true** enables the execution of the loop body which is given as a non terminal **Statement**.
- Statement, given as a non terminal **DesignatorStatement**, is executed at the end of the loop body except if the loop is terminated with the **break** statement.

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

- The numbers of actual and formal parameters must be the same.
- The type of every actual parameter must be assignment compatible with the type of every formal parameter at corresponding positions.

Condition = CondTerm {″||″ CondTerm}

CondTerm = CondFact {″&&″ CondFact}.

Condition = Expr Relop Expr.

- The types of both expressions must be compatible.
- Classes and arrays can only be checked for equality or inequality.

Expr = Term.

Expr = "-"Term.

- Term must be of type **int**.

Expr = Expr Addop Term.

- Expr and Term must be of type **int**.
- If the **Addop** is a combined arithmetic operator(+=,-=), **Expr** must denote a variable, array element or a field within an object.

Term = Factor.

Term = Term Mulop Factor.

- Term and Factor must be of type **int.**
- If the **Mulop** is a combined arithmetic operator(*=, /=, %=), **Expr** must denote a variable, array element or a field within an object.

Factor = Designator | number | charConst| "(" Expr ")".

Factor = Designator "(" ActPars ")".

- Designator must denote a static or non static method or a global function of the main program.
- Static method can be invoked only with a class or an object which of a class.

Factor = "new" Type .

- **Type** must denote a class.

Factor = "new" Type "[" Expr "]".

- The type of **Expr** must be **int**.

Designator = Designator "." ident .

- The type of **Designator** must be a class.
- **ident** must be a field or a method of an object denoted with the non terminal **Designator**.

Designator = Designator "[" Expr "]".

- The type of **Designator** must be an array.
- The type of **Expr** must be **int**.

Assignop = ”=” | AddopRight | MulopRight.

- Assignment operator is right associative.

Addop = AddopLeft | AddopRight. Mulop = MulopLeft | MulopRight.

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

AddopLeft = ″+″ | ″-″.

- Operators are left associative.

AddopRight = ″+=″ | ″-=″.

- Operators are right associative.

MulopLeft = ″*″ | ″/″ | ″%″.

- Operators are left associative.

MulopRight= ″*=″ | ″/=″ | ″%=″.

- Operators are right associative.

7.4 Implementation Restrictions

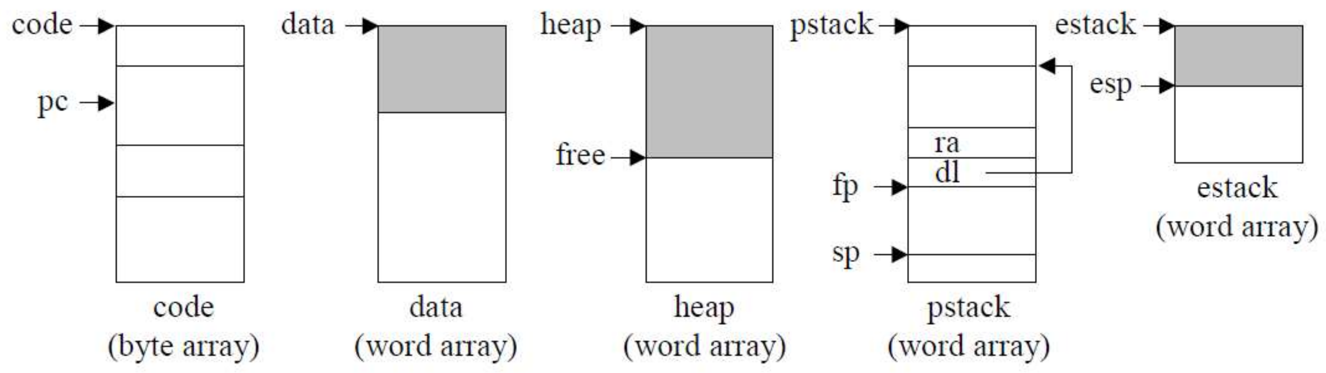
- There must not be more than 256 local variables.
- There must not be more than 65536 global variables.
- A class must not have more than 65536 fields.
- Source code of the program cannot exceed 8KB.

8. The MicroJava VM

The MicroJava VM is similar to the Java VM but has less and simpler instructions. Whereas the Java VM uses operand names from the constant pool that are resolved by the loader, the MicroJava VM uses fixed operand addresses. Java instructions encode the types of their operands so that a verifyer can check the consistency of an object file. MicroJava instructions do not encode operand types.

8.1 Memory Layout

The memory areas of the MicroJava VM are as follows.



code

This area contains the code of the methods. The register **pc** contains the index of the currently executed instruction. Register **mainpc** contains the start address of the method **main()**.

data

This area holds the (static or global) data of the main program. It is an array of variables. Every variable holds a single word (32 bits). The addresses of the variables are indexes into the array.

heap

This area holds the dynamically allocated objects and arrays. The blocks are allocated consecutively. Pointer **free** points to the beginning of the still unused area of the heap. Dynamically allocated memory is only returned at the end of the program. There is no garbage collector. All object fields hold a single word (32 bits). Arrays of char elements are byte arrays. Their length is a multiple of 4. Pointers are word offsets into the heap. Array objects start with an invisible word, containing the array length.

pstack

This area (the procedure stack) maintains the activation frames of the invoked methods. Every frame consists of an array of local variables, each holding a single word (32 bits). Their addresses are indexes into the array. **ra** is the return address of the method, **dl** is the dynamic link (a pointer to the frame of the caller). A newly allocated frame is initialized with all zeroes.

estack

This area (the expression stack) is used to store the operands of the instructions. After every MicroJava statement estack is empty. Method parameters are passed on the expression stack and are removed by the **enter** instruction of the invoked method. The expression stack is also used to pass the return value of the method back to the caller.

All data (global variables, local variables, heap variables) are initialized with a null value (0 for int, chr(0) for char, null for references).

8.2 Instruction Set

The following tables show the instructions of the MicroJava VM together with their encoding and their behaviour. The third column of the tables show the contents of estack before and after every instruction, for example

..., val, val

..., val

means that this instruction removes two words from estack and pushes a new word onto it. The operands of the instructions have the following meaning:

b a byte

a short int (16 bits)

w a word (32 bits)

Variables of type **char** are stored in the lowest byte of a word and are manipulated with word instructions (e.g. **load**, **store**). Array elements of type **char** are stored in a byte array and are loaded and stored with special instructions.

8.2.1 Loading and storing of local variables

Operation code	Instruction	Operands	estack	Meaning
1	load	b	... ..., val	Load push(local[b]);
2..5	load_n		... ..., val	Load (n = 0..3) push(local[n]);
6	store	b	..., val ...	Store local[b] = pop();
7..10	store_n		..., val ...	Store (n = 0..3) local[n] = pop();

8.2.2 Loading and storing of global variables

Operation code	Instruction	Operands	estack	Meaning
11	getstatic	s	... ..., val	Load static variable push(data[s]);
12	putstatic	s	..., val ...	Store static variable data[s] = pop();

8.2.3 Loading and storing of object fields

Operation code	Instruction	Operands	estack	Meaning

13	getfield	s	..., adr
			..., val
				Load object field
				adr = pop() / 4;
				push(heap[adr+s]);

14	putfield	s	..., adr, val
			...
				Store object field
				val = pop();
				adr = pop() / 4;
				heap[adr + s] = val;

8.2.4 Loading of constants

Operation code	Instruction	Operands	estack	Meaning
15..20	const_n		… …, val	Load constant (n = 0..5) push(n);
21	const_m1		… …, -1	Load minus one push(-1);
22	const	w	… …, val	Load constant push(w);

8.2.5 Arithmetic

Operation code	Instruction	Operands	estack	Meaning
23	add		…, val1, val2 …, val1 + val2	Add push(pop() + pop());
24	sub		…, val1, val2 …, val1 - val2	Subtract push(-pop() + pop());
25	mul		…, val1, val2 …, val1 * val2	Multiply push(pop() * pop());

26	div		…, val1, val2
			…, val1 / val2
				Divide
				x = pop();
				push(pop() / x);

27	rem		…, val1, val2
			…, val1 % val2
				Remainder
				x = pop();
				push(pop() % x);

28	neg		…, val1 …, -val1	Negate push(-pop());

29	shl		…, val, x
			…, val1
				Shift left
				x = pop();
				push(pop() << x);

30	Shr		…, val, x
			…, val1
				Shift right (arithmetically)
				x = pop();
				push(pop() >> x);

31	inc	b1, b2	… …	<u>Incrementing</u> local[b1] = local[b1] + 2;

8.2.6 Object creation

Operation code	Instruction	Operands	estack	Meaning
32	new	s	... ..., adr	New object allocate area of s words; initialize area to all 0; push(adr(area));

33	newarray	b	..., n
			..., adr
				New array
				n = pop();
				if (b==0)
				    alloc. array with n elems
				    of byte size;
				else if (b==1)
				    alloc. array with n elems
				    of word size;
				initialize array to all 0;
				push(adr(array))

8.2.7 Array access

Operation code	Instruction	Operands	estack	Meaning

34	aload		..., adr, ind
			..., val
				Load array element
				ind = pop();
				adr = pop() / 4 + 1;
				push(heap[adr + ind]);

35	astore		..., adr, ind, val
			...
				Store array element
				val = pop();
				ind = pop();
				adr = pop() / 4 + 1;
				heap[adr + ind] = val;

36	baload		..., adr, ind
			..., val
				Load byte array element
				ind = pop();
				adr = pop() / 4 + 1;
				x = heap[adr + ind / 4];
				push(byte ind % 4 of x);

37	bastore		..., adr, ind, val
			...
				Store byte array element
				val = pop();
				ind = pop();
				adr = pop() / 4 + 1;
				x = heap[adr + ind / 4];
				set byte ind % 4 in x;
				heap[adr + ind / 4] = x;

38	arraylength		..., adr
			..., len
				Get array length
				adr = pop();
				push(heap[adr]);

8.2.8 Stack manipulation

Operation code	Instruction	Operands	estack	Meaning
39	pop		..., val ..	Remove topmost stack element dummy = pop();

40	dup		..., val
			..., val, val
				Duplicate top element of the stack
				x = pop();
				push(x);
				push(x);

41	dup2		..., v1, v2
			..., v1, v2, v1, v2
				Duplicate top two elements of the stack
				y = pop();
				x = pop();
				push(x);
				push(y);
				push(x);
				push(x);

8.2.9 Jumps

The destination of the jump is relative to the start of the jump instruction.

Operation code	Instruction	Operands	estack	Meaning
42	jmp	s		Uncoditional jump pc = pc + s;

43..48	j<cond>	s	..., val1, val2
			...
				Conditional jump
				Supported conditions are:
				eq – equal
				ne – not equal
				lt – less than
				le – less than or equal
				gt – greater than
				ge – greater than or equal
				y = pop();
				x = pop();
				if (x cond y) pc = pc + s;

8.2.10 Method call

PUSH and POP work on *pstack*

Operation code	Instruction	Operands	estack	Meaning

49	call	s		Call method
				PUSH(pc + 3);
				pc = pc + s;

50	return			Return pc = POP();

51	enter	b1, b2		Enter method
				psize = b1; // in words
				lsize = b2; // in words
				PUSH(fp);
				fp = sp;
				sp = sp + lsize;
				initialize frame to 0;
				for (i = psize - 1; i >= 0; i--)
				    local[i] = pop();

52	exit			Exit method
				sp = fp;
				fp = POP();

8.2.11 Input/Output

Operation code	Instruction	Operands	estack	Meaning

53	read		...
			..., val
				Read
				readInt(x);
				push(x);

54	print		..., val, width
			...
				Print
				width = pop();
				writeInt(pop(), width);

55	bread		...
			..., val
				Read byte
				readChar(ch);
				push(ch);

56	bprint		..., val, width
			...
				Print byte
				width = pop();
				writeChar(pop(), width);

8.2.12 Miscellaneous

Operation code	Instruction	Operands	estack	Meaning
57	trap	b		Generate run time error print error message depending on b;

				stop execution;
58	invokevirtual	$w_1, w_2, \ldots w_n, w_{n+1}$	..., adr ...	Virutal method call the name of the method has n characters; these characters are part of the instruction and are located in words $w_1, w_2, \ldots w_n$. word w_{n+1} has a value of -1 and depicts the end of the instruction; instruction first removes the adr from the expression stack; adr is the address in the static memory zone and it points to virtual function table of the class of the object for which the method is called. If the name of the method is found within the virtual function table the instruction jumps to the method body.

8.2.13 Combined operators

Operation code	Instruction	Operands	estack	Meaning
59	dup_x1		..., v1, v2 ..., v2, v1, v2	Inserting a copy of the top stack elemnt bellow the second stack element
60	dup_x2		..., v1, v2, v3 ..., v3, v1, v2, v3	Inserting a copy of the top stack elemnt bellow the third stack element

8.3 Object file format

2 bytes: "MJ"

4 bytes: code size in bytes

4 bytes: number of words for the global data

4 bytes: mainPC: the address of main() relative to the beginning of the code area

n bytes: the code area (n = code size specified in the header)

8.4 Run Time Errors

1	Missing return statement in a function.