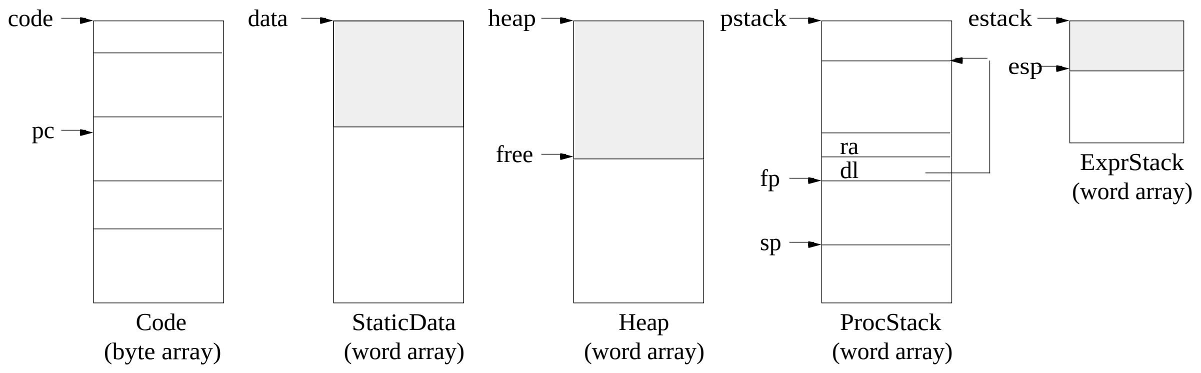
Appendix B. The MicroJava VM

This section describes the architecture of the MicroJava Virtual Machine that is used in the practical part of this compiler construction course. The MicroJava VM is similar to the Java VM but has less instructions. Some instructions were also simplified. Where 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.

B.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. *mainpc* contains the start address of the method *main()*.
StaticData	This area holds the (static or global) data of the main program (i.e. of the class to be compiled). It is an array of variables. Every variable occupies one 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. *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 occupy a single word (32 bits). Arrays of *char* elements are byte arrays. Their length is a multiple of 4. Pointers are byte offsets into the heap. Array objects start with an invisible word, containing the array length.
ProcStack	In this area the VM maintains the activation frames of the invoked methods. Every frame consists of an array of local variables, each occupying 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.
ExprStack	This area is used to store the operands of the instructions. After every MicroJava statement *ExprStack* 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).

B.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 ExprStack before and after every instruction, for example

…, val, val
…, val

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

b	a byte
s	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.

Loading and storing of local variables

opcode	***instr.***	*opds*	*ExprStack*	*meaning*
1	**load**	b	…	Load
			…, val	push(local[b]);
2..5	**load_n**		…	Load (n = 0..3)
			…, val	push(local[n]);
6	**store**	b	…, val	Store
			…	local[b] = pop();
7..10	**store_n**		…, val	Store (n = 0..3)
			…	local[n] = pop();

Loading and storing of global variables

11	**getstatic**	s	…	Load static variable
			…, val	push(data[s]);
12	**putstatic**	s	…, val	Store static variable
			…	data[s] = pop();

Loading and storing of object fields

13	**getfield**	s	…, adr	Load object field
			…, val	adr = pop()/4; push(heap[adr+s]);
14	**putfield**	s	…, adr, val	Store object field
			…	val = pop(); adr = pop()/4;
				heap[adr+s] = val;

Loading of constants

15..20	**const_n**		…	Load constant (n = 0..5)
			…, val	push(n)
21	**const_m1**		…	Load minus one
			…, -1	push(-1)
22	**const**	w	…	Load constant
			…, val	push(w)

Arithmetic

23	**add**		…, val1, val2	Add
			…, val1+val2	push(pop() + pop());
24	**sub**		…, val1, val2	Subtract
			…, val1-val2	push(-pop() + pop());
25	**mul**		…, val1, val2	Multiply
			…, val1*val2	push(pop() * pop());
26	**div**		…, val1, val2	Divide
			…, val1 / val2	x = pop(); push(pop() / x);
27	**rem**		…, val1, val2	Remainder
			…, val1%val2	x = pop(); push(pop() % x);
28	**neg**		…, val	Negate
			…, - val	push(-pop());
29	**shl**		…, val	Shift left
			…, val1	x = pop(); push(pop() << x);
30	**shr**		…, val	Shift right (arithmetically)
			…, val1	x = pop(); push(pop() >> x);
31	**inc**	b1, b2	…	Increment variable
			…	local[b1] = local[b1] + b2;

Object creation

32	**new**	s	…	New object
			…, adr	allocate area of s bytes;
				initialize area to all 0;
				push(adr(area));
33	**newarray**	b	…, n	New array
			…, adr	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))

Array access

34	**aload**	…, adr, index
		…, val
			Load array element (+ index check)
			i = pop(); adr = pop()/4+1;
			push(heap[adr+i]);

35	**astore**	…, adr, index, val
		…
			Store array element (+ index check)
			val = pop(); i = pop(); adr = pop()/4+1;
			heap[adr+i] = val;

36	**baload**	…, adr, index
		…, val
			Load byte array element (+ index check)
			i = pop(); adr = pop()/4+1;
			x = heap[adr+i/4];
			push(byte i%4 of x);

37	**bastore**	…, adr, index, val
		…
			Store byte array element (+ index check)
			val = pop(); i = pop(); adr = pop()/4+1;
			x = heap[adr+i/4];
			set byte i%4 in x;
			heap[adr+i/4] = x;

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

Stack manipulation

39	**pop**	…, val …	Remove topmost stack element dummy = pop();
40	**dup**	…, val …, val, val	Duplicate topmost stack element x = pop(); push(x); push(x);

41	**dup2**	…, v1, v2
		…, v1, v2, v1, v2
			Duplicate top two stack elements
			y = pop(); x = pop();
			push(x); push(y); push(x); push(y);

Jumps

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

42	**jmp** s		Jump unconditionally pc = pc + s;

43..48	j**<cond>** s	…, val1, val2
		…
			Jump conditionally (eq, ne, lt, le, gt, ge)
			y = pop(); x = pop();
			if (x cond y) pc = pc + s;

Method call (PUSH and POP work on the procedure stack)

49	**call**	s		Call method
				PUSH(pc+3); pc := pc + s;
50	**return**			Return
				pc = POP();
51	**enter**	b1, b2		Enter method
				psize = b1; 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();

Input/Output

53	**read**		…	Read
			…, val	readInt(x); push(x);
				// reads from standard input
54	**print**		…, val, width	Print
			…	width = pop(); writeInt(pop(), width);
				// writes to standard output
55	**bread**		…	Read byte
			…, val	readChar(ch); push(ch);
56	**bprint**		…, val, width	Print byte
			…	width = pop(); writeChar(pop(), width);

Other

57	**trap**	b		Generate run time error
				print error message depending on b;
				stop execution;

B.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)

B.4 Run Time Errors

1 Missing return statement in function.