Appendix A. The MicroJava Language

This section describes the MicroJava language that is used in the practical part of the compiler construction module. MicroJava is similar to Java but much simpler.

A.1 General Characteristics

- A MicroJava program consists of a single program file with static fields and static methods. There are no external classes but only inner classes that can be used as data types.
- The main method of a MicroJava program is always called *main()*. When a MicroJava program is called this method is executed.
- There are
  - Constants of type *int* (e.g. 3) and *char* (e.g. 'x') but no string constants.
  - Variables: all variables of the program are static.
  - Primitive types: *int*, *char* (Ascii)
  - Reference types: one-dimensional arrays like in Java as well as classes with fields but without methods.
  - Static methods in the main class.
- There is no garbage collector (allocated objects are only deallocated when the program ends).
- Predeclared procedures are *ord*, *chr*, *len*.

Sample program

```
program P
  final int size = 10;

  class Table {
    int[] pos;
    int[] neg;
  }

  Table val;

{
  void main()
    int x, i;
  { //---------- Initialize val ------------
    val = new Table;
    val.pos = new int[size];
    val.neg = new int[size];
    i = 0;
    while (i < size) {
      val.pos[i] = 0; val.neg[i] = 0;
      i = i + 1;
    }
    //------------ Read values -------------
    read(x);
    while (x != 0) {
      if (x >= 0) {
        val.pos[x] = val.pos[x] + 1;
      } else if (x < 0) {
        val.neg[-x] = val.neg[-x] + 1;
      }
      read(x);
    }
  }
}
```

A.2 Syntax

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

ConstDecl   = "final" Type ident "=" (number | charConst) ";".
VarDecl     = Type ident {"," ident } ";".
ClassDecl   = "class" ident "{" {VarDecl} "}".
MethodDecl  = (Type | "void") ident "(" [FormPars] ")" {VarDecl} Block.
FormPars    = Type ident  {"," Type ident}.
Type        = ident ["[" "]"].

Block       = "{" {Statement} "}".
Statement   = Designator ("=" Expr | ActPars) ";"
            | "if" "(" Condition ")" Statement ["else" Statement]
            | "while" "(" Condition ")" Statement
            | "return" [Expr] ";"
            | "read" "(" Designator ")" ";"
            | "print" "(" Expr ["," number] ")" ";"
            | Block
            | ";".
ActPars     = "(" [ Expr {"," Expr} ] ")".

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

Expr        = ["-"] Term {Addop Term}.
Term        = Factor {Mulop Factor}.
Factor      = Designator [ActPars]
            | number
            | charConst
            | "new" ident ["[" Expr "]"]
            | "(" Expr ")".
Designator  = ident {"." ident | "[" Expr "]"}.
Addop       = "+" | "-".
Mulop  =      "*" | "/" | "%".
```

Lexical structure

Character classes:

```
letter     = 'a'..'z' | 'A'..'Z'.
digit      = '0'..'9'.
whiteSpace = ' ' | '\t' | '\r' | '\n'.
```

Terminal classes:

```
ident      = letter {letter | digit}.
number     = digit {digit}.
charConst  = "'" char "'".  // including '\r', '\t', '\n'
```

Keywords:

```
program class
if      else    while   read    print   return
void    final   new
```

Operators:

```
+       -       *       /       %
==      !=      >       >=      <       <=
(       )       [       ]       {       }
=       ;       ,       .
```

Comments:

```
// to the end of line
```

A.3 Semantics

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

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.

Same type
Two types are the same
- 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 *dst* are the same, or
- if *dst* is a reference type and *src* is the type of *null*.

Predeclared names

int	the type of all integer values
char	the type of all character values
null	the null value of a class or array variable, meaning "pointing to no value"
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 method; *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 class 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).

A.4 Context Conditions

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.

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*.

Context conditions for the MicroJava productions

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

ConstDecl = "final" Type ident "=" (number | charConst) ";".

- The type of *number* or *charConst* must be the same as the type of *Type*.

VarDecl = Type ident {"," ident } ";".

ClassDecl = "class" ident "{" {VarDecl} "}".

MethodDecl = (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).

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

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

- *ident* must denote a type.

Statement = Designator "=" 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*.

Statement = Designator ActPars ";".

- *Designator* must denote a method.

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

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

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

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

Statement = "return" [Expr] .

- The type of *Expr* must be assignment compatible with the function type of the current method.
- If *Expr* is missing the current method must be declared as void.

Statement = "if" "(" Condition ")" Statement ["else" Statement]
| "while" "(" Condition ")" Statement
| "{" {Statement} "}"
| ";".

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

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

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*.

Term = Factor.

Term = Term Mulop Factor.

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

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

Factor = Designator ActPars.

- *Designator* must denote a method.

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 of *Designator*.

Designator = Designator "[" Expr "]".

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

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

Addop = "+" | "-".

Mulop = "*" | "/" | "%".

A.5 Implementation Restrictions

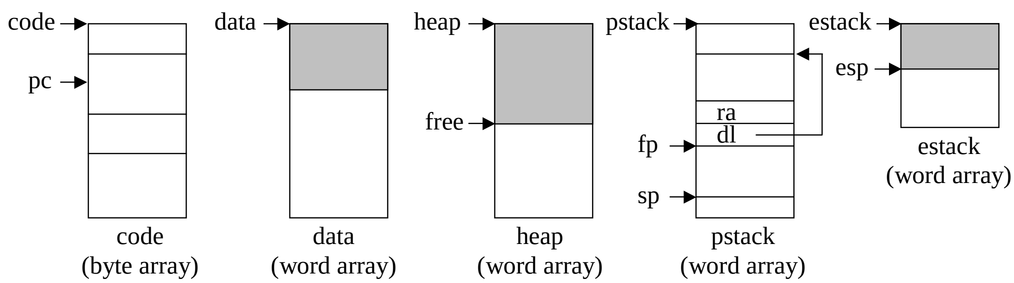
- There must not be more than 127 local variables.
- There must not be more than 32767 global variables.
- A class must not have more than 32767 fields.

Appendix B. The MicroJava VM

This section describes the architecture of the MicroJava Virtual Machine that is used in compiler lab. 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.

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*().

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. *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 byte 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).

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 *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
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

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();

Loading and storing of global variables

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

Loading and storing of object fields

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;

Loading of constants

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);

Arithmetic

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**	..., val ..., - val	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);

Object creation

31	**new** s	... ..., adr	New object allocate area of s bytes; initialize area to all 0; push(adr(area));

32	**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))

Array access

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

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

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

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

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

Stack manipulation

38	**pop**	..., val ...	Remove topmost stack element dummy = pop();

Jumps

39	**jmp** s		Jump unconditionally pc = s;

40..45	**j<cond>** s	..., x, y
		...
			Jump conditionally (eq, ne, lt, le, gt, ge)
			y = pop(); x = pop();
			if (x cond y) pc = s;

Method call (PUSH and POP work on *pstack*)

46	**call** s		<u>Call method</u> PUSH(pc+3); pc = s;
47	**return**		<u>Return</u> pc = POP();

48	**enter** b1, b2		<u>Enter method</u>
			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();

49	**exit**		<u>Exit method</u> sp = fp; fp = POP();

Input/Output

50	**read**	... ..., val	<u>Read</u> readInt(x); push(x);
51	**print**	..., val, width ...	<u>Print</u> width = pop(); writeInt(pop(), width);
52	**bread**	... ..., val	<u>Read byte</u> readChar(ch); push(ch);
53	**bprint**	..., val, width ...	<u>Print byte</u> width = pop(); writeChar(pop(), width);

Miscellaneous

54	**trap** b		<u>Generate run time error</u> 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 a function.