Appendix A. The MicroJava Language – Extended

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

A.1 General Characteristics

- A MicroJava programme consists of a single class with static fields and static methods. There are no other 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 programme is called this method is executed.
- There are
  - Constants of type *int* and *char* but no string constants.
  - Variables: all variables contain references (pointers); variables in the main class are static.
  - Basic types: *int*, *char* (Ascii).
  - Structured types: onedim. Arrays like in Java, inner classes with fields and dynamic methods.
  - Static methods in the main class.
  - Dynamic methods in inner classes. There are no constructors.
- Every inner class can inherit from some other inner class, except from itself. MicroJava is a single-inheritance language.
- Inheritance in MicroJava has the same principles as in Java. Subclass object is also a type of its superclass. The subclass reference can be assigned to any of its superclass references. The act of converting a subclass reference into a superclass reference is called *upcasting*.
- Superclass methods can be overridden in its subclass. Because of that, inner class methods binding occurs at run time, based on the type of object. This is polymorphism (also called *dynamic binding* or *late binding* or *run-time binding*).
- Within an instance method, the name of an instance variable refers to the current object's instance variable, assuming that the instance variable isn't hidden by a method parameter. If it is hidden, we can access instance variable through this.variableName. In the body of inner methods this represents reference to the object that invoked method.
- Method overloading is not supported in MicroJava.
- There is no Object class, the top-most class, the class from which all other classes are implicitly derived.
- There is no garbage collector (allocated objects are only deallocated when the programme ends).
- Predeclared static methods are *ord, chr, len.*

Sample programme (keywords are underlined)

```
class DeliveryShop

   final int SIZE = 100;

   final int DEADLINE = 7;

   final char ANSWERP = 'Y';

   final char ANSWERN = 'N';
```

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

{
    void initialize ()
          int i;
    {  // ---------- Initialize Table object
          pos = new int[SIZE]; neg = new int[SIZE];
          i = 0;
          while (i < SIZE) {
               pos[i] = 0; neg[i] = 0;
               i++;
          }
    }

    void setRating (int time)
          int x;
    {
     read(x);
     if (x >= 0 && x <= SIZE)
          if (time >=0 && time <= DEADLINE) pos[x] = pos[x] + 5;
          else if (time < 2*DEADLINE) neg[x] = -3;
               else neg[x] = -15;
     }

      char permission ()
          int s, j;
      {
       s=0; j=0;
       while (j < SIZE) { s = s + pos[j] + neg[j]; j++; }
        if  (s < 0) return ANSWERN;
        else return ANSWERP;
        }

}
}

class Box {
    int weight;
    int width;
    int height;
    int depth;
    int unitPrice;
{
    void Box1 (int weight, int width, int height, int depth, int unitPrice)
    {
     this.weight = weight;
      this.width = width;
      this.height = height;
      this.depth = depth;
```

```
        this.unitPrice = unitPrice;
        }

        int volume ()
        {
         return weight*width*height;
         }

          int totalPrice()
          {
          return weight*unitPrice + 10;
           }
  }
  }

  class GiftBox extends Box {
      int kind;
  {
     void GiftBox1 (int we, int wi, int he, int de, int up, int k)
     {
      this.Box1(we, wi, he, de, up);
      kind = k;
       }

       int totalPrice ()    // overridden method
             int ad;
       {
        if (kind == 1) ad = 5;
        else if (kind == 2) ad = 10;
              else if (kind == 3) ad=15;
        return weight*unitPrice + 10 + ad;
        }
  }
  }

  class Customer {
       int idNum;
       Table val;
   }

{

Table tableConstructor ()   // static method
   Table t;
{
t = new Table;
return t;
}

void main ()
   Customer John;
```

```
   Box b;
   GiftBox gb;
   int bill;
   int time;
{
John = new Customer;
John.idNum = 0;
John.val = tableConstructor();
John.val.initialize();
gb = new GiftBox;
gb.GiftBox1(2, 500, 500, 500, 40, 2);
b = gb;      // upcasting
if (John.val.permission() == ANSWERP) bill = b.totalPrice();   // polymorphism
else { print('E');
        print('R');
        print('O');
        print('R'); }
read(time);
John.val.setRating(time);
}

}
```

A.2 Syntax

```
   Program       = "class" ident {ConstDecl | VarDecl | ClassDecl} "{" {MethodDecl} "}".
   ConstDec1     = "final" Type ident "=" (number | charConst) ";".
   VarDec1       = Type ident ["[" "]"] {"," ident ["[" "]"]} ";".
   ClassDec1     = "class" ident ["extends"] "{" {VarDecl} ["{" {MethodDec1} "}"] "}".
   MethodDec1    = (Type | "void") ident "(" [FormPars] ")" {VarDecl} "{" {Statement} "}".
   FormPars      = Type ident ["[" "]"] {"," Type ident ["[" "]"]}.
   Type          = ident.
   Statement     = Designator ("=" Expr | "(" [ActPars] ")" | "++" | "--") ";"
                 | "if" "(" Condition ")" Statement ["else" Statement]
                 | "while" "(" Condition ")" Statement
                 | "break" ";"
                 | "return" [Expr] ";"
                 | "read" "(" Designator ")" ";"
                 | "print" "(" Expr ["," number]  ")" ";"
                 | "{" {Statement} "}".
   ActPars       = Expr {"," Expr}.
   Condition     = CondTerm {"||" CondTerm}.
   CondTerm      = CondFact {"&&" CondFact}.
   CondFact      = Expr Relop Expr.
   Expr          = ["-"] Term {Addop Term}.
   Term          = Factor {Mulop Factor}.
   Factor        = Designator ["(" [ActPars] ")"]
                 | number
                 | charConst
                 | "new" Type ["[" Expr "]"]
```

```
                | "(" Expr ")".
Designator      = ident {"." ident | "[" Expr "]"}.
Relop           = "==" | "!=" | ">" | ">=" | "<" | "<=".
Addop           = "+" | "-".
Mulop           = "*" | "/" | "%".
```

Lexical Structure

Keywords:	break, class, else, extends, final, if, new, print, read, return, void, while
Token classes:	ident = letter {letter | digit | "_"}.
	number = digit {digit}.
	charConst = "'" printableChar "'".
Operators:	+, -, *, /, %, ==, !=, >, >=, <, <=, &&, ||, =, ++, --, ;, comma, . (,), [,], {, }
Comments:	// to the end of the 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*, or
- if *dst* is a superclass reference and *src* is a subclass reference.

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"

eol	end of line character (corresponds to '\n'); print(eol) skips to the next output 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 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, to which the main class is local and which contains all predeclared names.

The declaration of a name in an inner scope *S* hides the declarations of the same name in outer scopes.

Note

- Indirect recursion is 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).
- Name this is not predeclared, or it is keyword, but it is not recommended to use it, except in dynamic methods as it is explained.

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 programme must contain a method named *main*. It must be declared with a void function type 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 MicroJava productions

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

ConstDec1 = "final" Type ident "=" (number | charConst) ";".

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

VarDec1 = Type ident ["[" "]"] {"," ident ["[" "]"]} ";".

ClassDec1 = "class" ident ["extends"] "{" {VarDecl} ["{" {MethodDec1} "}"] "}".

MethodDec1 = (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 ("++" | "--") ";".
- *Designator* must denote a variable, an array element or an object field.
- *Designator* must be of type int.

Statement = Designator "(" [ActPars] ")" ";".
- *Designator* must denote a method.

Statement = "break".
- The break statement must be contained in a while statement.

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 = CondTerm {"||" CondTerm}.

CondTerm = CondFact {"&&" CondFact}.

CondFact = 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 or method 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

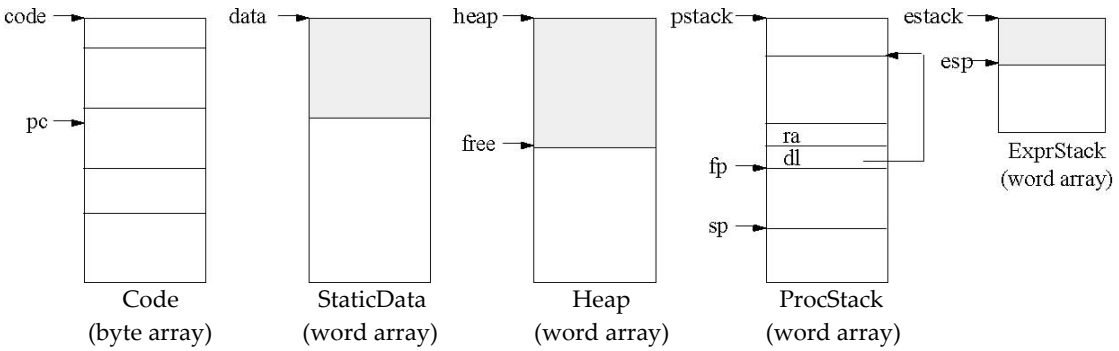
- There must not be more than 256 local variables.
- There must not be more than 65536 global variables.
- A class must not have more than 65536 class members (fields and methods).
- The code of the program must not be longer than 8 KBytes.

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 instruction. 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 verifier 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. This area also holds method tables (virtual tables) for all inner classes.
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). First field, on position 0, of every object contains the start address of the method table for that object in StaticData. 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 ..., 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 ..., val1	Shift left x = pop(); push(pop() << x);
30	**shr**		..., val ..., val1	Shift right (arithmetically) 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));
				e.g. if class has two fields, then new 8,
				because every field occupies one word
33	**newarray**	b	..., n	New array
			..., adr	n is array length
				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	Load array element (+ index check)
		..., val	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	Load byte array element (+ index check)
		..., val	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	Get array length
		..., len	adr = pop();
			push(heap[adr]);

Stack manipulation

39	**pop**	..., val	Remove topmost stack element
		...	dummy = pop();
40	**dup**	..., val	Duplicate topmost stack element
		..., val, val	x = pop(); push(x); push(x);
41	**dup2**	..., v1, v2	Duplicate top two stack elements
		..., v1, v2, v1, v2	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 static method PUSH(pc+3); pc = pc + s;
58	**invokevirtual**	w_1, w_2, w_3, ..., w_n,w_{n+1}	..., adr ...	Call dynamic method method name has n characters; these characters are stored in the instruction itself, in words w_1..w_n; word w_{n+1} is equal -1, and marks the end of the instruction; instruction first removes adr from expression stack; adr is address in the static data where starts method table for object that invoked method; if name in instruction matches with one of the names in table instruction provides jump to method body;
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**		..., val ...	Read readInt(x); push(x); *// reads from standard input*
54	**print**		..., val, width ...	Print width = pop(); writeInt(pop(), width); *// writes to standard output*
55	**bread**		... ..., val	Read byte 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 n bytes: the code area (n = code size specified

B.4 Run Time Errors

1 Missing return statement in function.