Handouts

Compiler Construction

Prof. Dr. Hanspeter Mössenböck
Johannes Kepler University Linz

hanspeter.moessenboeck@jku.at

2016

http://ssw.jku.at/Misc/CC/

Course Contents

1. Overview
   1.1 Motivation
   1.2 Structure of a compiler
   1.3 Grammars
   1.4 Chomsky's classification of grammars
   1.5 The MicroJava language

2. Scanning
   2.1 Tasks of a scanner
   2.2 Regular grammars and finite automata
   2.3 Scanner implementation

3. Parsing
   3.1 Context-free grammars and push-down automata
   3.2 Recursive descent parsing
   3.3 LL(1) property
   3.4 Error handling

4. Semantic processing and attribute grammars

5. Symbol table
   5.1 Overview
   5.2 Objects
   5.3 Scopes
   5.4 Types
   5.5 Universe

6. Code generation
   6.1 Overview
   6.2 The MicroJava VM
   6.3 Code buffer
   6.4 Operands
   6.5 Expressions
   6.6 Assignments
   6.7 Jumps
   6.8 Control structures
   6.9 Methods

7. Building generators with Coco/R
   7.1 Overview
   7.2 Scanner specification
   7.3 Parser specification
   7.4 Error handling
   7.5 LL(1) conflicts
   7.6 Example

Main Literature

- *N. Wirth: Compiler Construction. Addison-Wesley 1996*
  A master of compiler constructions teaches how to write simple and efficient compilers.
  Also available under http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf
- P.Terry: *Compiling with C# and Java*. Pearson Addison-Wesley 2005
  A very good book that covers most of the topics of this course. It also describes automatic compiler generation using the compiler generator Coco/R.
- *A.W.Appel: Modern Compiler Implementation in Java. Cambridge University Press 1998*
  Good and up-to-date book that treats the whole area of compiler construction in depth.
- *H. Mössenböck: The Compiler Generator Coco/R. User Manual.*
  http://ssw.jku.at/Coco/Doc/UserManual.pdf

Further Reading

- *S.Muchnick: Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.*
  A very good and complete book which goes far beyond the scope of this introductory course. Not quite cheap but rewarding if you really want to become a compiler expert.
- *H.Bal, D.Grune, C.Jacobs: Modern Compiler Design. John Wiley, 2000*
  Also a good books that describes the state of the art in compiler construction.
- *Aho, R. Sethi, J. Ullman: Compilers –Principles, Techniques and Tools. Addison-Wesley, 1986.*
  Old but still good to read. Does not cover recursive descent compilation in depth but has chapters about optimisation and data flow analysis.
- *W.M.Waite, G.Goos: Compiler Construction. Springer-Verlag 1984*
  Theoretical book on compiler construction. Good chapter on attribute grammars.

Compiler Construction Lab

In this lab you will write a small compiler for a Java-like language (*MicroJava*). You will learn how to put the presented techniques into practice and will study all the details involved in the implementation of a "real" compiler.

The project consists of four levels:

- Level 1 requires you to implement a scanner and a parser for the language MicroJava, specified in Appendix A of this document.
- Level 2 deals with symbol table handling and some type checking.
- If you want to go to full length with your compiler you should also implement level 3, which deals with code generation for the MicroJava Virtual Machine specified in Appendix B of this document. This level is (more or less) optional so that you can get a good mark even if you do not implement it. But you should try to do it.
- Level 4 finally requires you to use the compiler generator Coco/R to produce a compiler-like program automatically.

The marking scheme will be as follows:

class test	up to 45 points
project level 1	+ 20 points
project level 2	+ 20 points
project level 3	+ 5 points
project level 4	+ 10 points
	100 points

The project should be implemented in Java using Oracle's Java Development Kit (JDK, http://www.oracle.com/technetwork/java/javase/downloads/) or some other Java development environment.

Level 1: Scanning and Parsing

In this part of the project you will implement a scanner and a recursive descent parser for MicroJava. Start with the implementation of the scanner and do the following steps:

1. Study the specification of MicroJava carefully (Appendix A). What are the tokens of the MicroJava grammar? What is the syntax of identifiers, numbers, character constants and comments? What keywords and predeclared names do you need?
2. Create a package *MJ* and download the files *Scanner.java* and *Token.java* from http://ssw.jku.at/Misc/CC/ into this package. Look at those files and try to understand what they do.
3. Complete the skeleton file *Scanner.java* according to the slides of the course and compile *Token.java* and *Scanner.java*.
4. Download the file *TestScanner.java* into the package MJ and compile it.
5. Download the MicroJava program *sample.mj* and run *TestScanner* on it.
6. Download the MicroJava program *BuggyScannerInput.mj* and run *TestScanner* on it in order to check if incorrect tokens are handled properly.

Next, you should write a recursive descent parser that uses your scanner to read and check the input tokens. Do the following steps:

1. Download the file *Parser.java* into the package *MJ* and see what it does.
2. Complete the skeleton file *Parser.java* according to the slides of the course. Write a recursive descent parsing method for every production of the MicroJava grammar (see Appendix A). Compile *Parser.java*.
3. Download the file *TestParser.java*, compile it, and run it on *sample.mj*. If your parser is correct no errors should be reported.
4. Extend *Parser.java* with an error recovery according to the slides of the course. Add synchronisation points at the beginning of statements and declarations.
5. Download the MicroJava program *BuggyParserInput.mj* and run *TestParser* on it in order to check if syntax errors are handled correctly.

Level 2: Symbol Table Handling

Extend your parser with semantic processing. At every declaration the declared name must be entered into the symbol table. When a name occurs in a statement it must be looked up in the symbol table and the necessary context conditions must be checked (see Appendix A.4). Do the following steps:

1. Create a new package *MJ.SymTab* (i.e. a package *SymTab* within the package *MJ*).
2. Download the files *Obj.java*, *Struct.java*, *Scope.java* and *Tab.java* into this package and see what they do.
3. Complete the skeleton file *Tab.java* (i.e. the symbol table) according to the slides of the course.
4. Add semantic actions to *Parser.java*. These actions should enter every MicroJava name into the symbol table when it is declared and should retrieve it from the symbol table when it is used. The semantic actions should also open and close scopes appropriately. Try to check also some context conditions from Appendix A.4 (most context conditions can only be checked during code generation).
5. Compile everything and run *TestParser.java* on *sample.mj* again to see if it works. Insert some semantic errors into *sample.mj*. For example, use a few names without declaring them or declare some names twice and see if your compiler detects those errors.
6. In order to check whether you built the symbol table correctly you can call *Tab.dumpScope* whenever you have processed all declarations of a scope in your MicroJava program.

Level 3: Code Generation

The next task is to generate code for the MicroJava Virtual Machine. Before you start, carefully study the specification of the VM (Appendix B) in order to become familiar with the run-time data structures, the addressing modes, and the instructions. Then do the following steps:

1. Create a new package *MJ.CodeGen*.
2. Download the files *Code.java*, *Operand.java* and *Decoder.java* into this package.
3. Complete the skeleton file *Code.java* according to the slides of the course.
4. Add semantic actions to *Parser.java*. These actions should call the methods of *Code.java* and *Operand.java* as shown on the slides. Start with the actions for selectors (e.g. *obj.f* and *arr*[*i*]), and continue with the semantic actions for expressions, assignments, if statements, while statements and method calls. Note that most context conditions from Appendix A.4 have to be checked here as well.
5. Download the file *Compiler.java* into the package *MJ*. This is the main program of your compiler that replaces *TestParser.java*. Compile it and run it on *sample.mj*. This should produce a file *sample.obj* with the compiled program.
6. Download *BuggySemanticInput.mj* and check if your compiler detects all semantic errors in this MicroJava program.

In order to run your compiled MicroJava programs download the file *Run.java* (i.e. the MicroJava Virtual Machine) into the package *MJ* and compile it. You can invoke it with

```
java MJ.Run sample.obj [-debug]
```

You can also decode a compiled MicroJava program by downloading the file *Decode.java* to the package *MJ* and compiling it. You can invoke it with

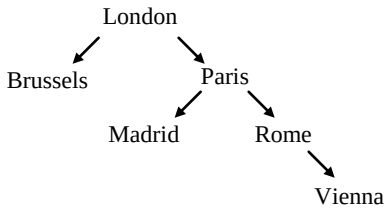
```
java MJ.Decode sample.obj
```

Level 4: The Compiler Generator Coco/R

This task requires you to use the compiler generator Coco/R for building programs that process structured input. It consists of two subtasks of which you have to implement at least one.

Task 1: Reading and Building a Binary Tree

Binary trees are dynamic data structures consisting of nodes, where every node has at most 2 sons, which are again binary trees. Assume that we want to build the following binary tree:



Data type for nodes

```
class Node {
    String name;
    Node left;
    Node right;
}
```

We want to read the tree from an input file, which represents the tree structure with parentheses, i.e.:

```
(London
    (Brussels)
    (Paris
        (Madrid)
        (Rome
            ()
            (Vienna)
        )
    )
)
```

Describe the input of such trees by a recursive EBNF grammar. Write a Coco/R compiler description using this grammar. Terminal symbols are identifiers as well as '(' and ')'. Add attributes and semantic actions to your compiler description in order to build the corresponding binary tree. Write also a dump method that prints the tree after it was built.

In order to use Coco/R go to http://ssw.jku.at/Coco/#Java and download the files *Coco.jar*, *Scanner.frame* and *Parser.frame* into a new directory *Tree*. If your compiler description is in a file *Tree.atg* in the directory *Tree* go to this directory and type

```
java -jar Coco.jar Tree.atg
```

This will generate the files *Scanner.java* and *Parser.java* in the directory *Tree*. Write a main program *TreeBuilder.java* that creates a scanner and a parser and calls the parser (look at the slides in the course).

Task 2: Building a Phone Book

Assume that we have a text file with phone book entries. Every entry consists of a person's name and one or more phone numbers for that person. A sample phone book might look like this:

```
Boulder, John M.
    home 020 7815 1234
    office 020 3465 234
Brown, Cynthia 1234567
Douglas, Ann Louise
    office +44 (0)20 234 567
    mobile +43 (0)664 7865 234
...
```

- Names consist of letters and may be abbreviated with a dot.
- Phone numbers consist of an optional country code (e.g. +44), an optional city code (e.g. 020) and a phone number consisting of one or several digit sequences. Country codes start with a '+' and must be followed by a city code (with a '0' in brackets). City codes without country codes start with a '0'. If there is no country code the default is +44. If there is no city code the default is 020.
- Phone numbers may be preceded by the words "home", "office" or "mobile". If such a word is missing the default is "home".

Describe the syntax of such a phone book file by a grammar. Write a Coco/R compiler description that processes such input files by reading them and building a phone book data structure in memory, where every entry of this data structure holds the family name, the first name(s), and the phone number(s) including the country code, the city code and the kind of phone number as separate fields. Write also a dump method that prints the whole phone book.

Appendix A. The MicroJava Language

This section describes the MicroJava language that is used in the labs 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 local 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.
- There is no garbage collector (objects are deallocated when the program ends).
- Predeclared methods 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

If a term in this document is underlined it refers to the following definitions:

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 method or class name in the declaration to the closing curly brace 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 program 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 in MicroJava, 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 predeclared 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" ident .

- The type of *ident* must be a class.

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

- *ident* must denote a type.
- 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 "]".

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

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

Addop = "+" | "-".

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

A.5 Implementation Restrictions

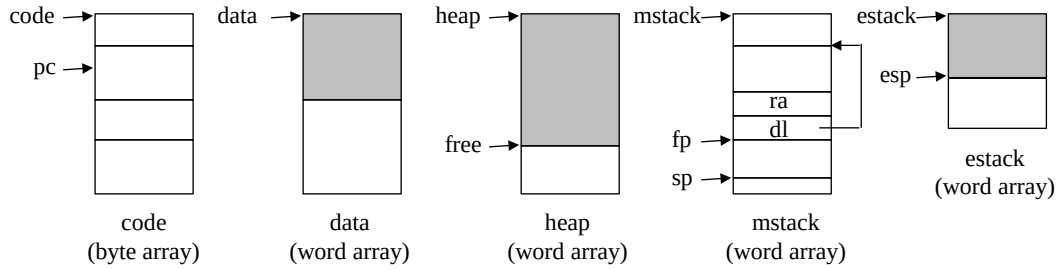
- There must not be more than 127 local variables.
- There must not be more than 32767 global variables.
- A class must not have more than 32767 fields.

Appendix B. The MicroJava VM

This section describes the architecture of the MicroJava Virtual Machine that is used in the 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. There is no garbage collector. Dynamically allocated memory is only returned at the end of the program. 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.

mstack — This area (the method 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 expressions. 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 shows 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 words; 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();
			push(heap[adr+1+i]);

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

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

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

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

48	**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();

49	**exit**		Exit method sp = fp; fp = POP();

Input/Output

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

Miscellaneous

54	**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 a function.