

## 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



# *Tasks of Code Generation*

## **Generation of machine instructions**

- selecting the right instructions
- selecting the right addressing modes

## **Translation of control structures (if, while, ...) into jumps**

## **Allocation of stack frames for local variables**

## **Maybe some optimizations**

## **Output of the object file**

# Common Strategy of Code Generation



## 1. Study the target machine

registers, data formats, addressing modes, instructions, instruction formats, ...

## 2. Design the run-time data structures

layout of stack frames, layout of the global data area, layout of heap objects, ...

## 3. Implement the code buffer

instruction encoding, instruction patching, ...

## 4. Implement register allocation

irrelevant in MicroJava, because we have a stack machine

## 5. Implement code generation routines (in the following order)

- load values (to the expression stack)
- process designators (x.y, a[i], ...)
- translate expressions
- manage labels and jumps
- translate statements
- translate methods and parameter passing

## 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

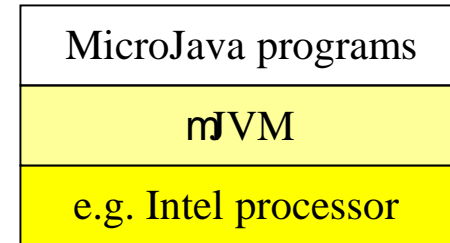
6.9 Methods

# Architecture of the MicroJava VM (mJVM)



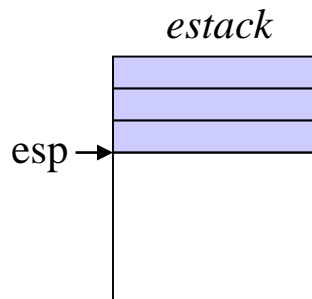
## What is a virtual machine (VM)?

- A software CPU
- instructions are interpreted (or "jitted")
- examples: Java VM, Smalltalk VM, Pascal P-Code



## The mJVM is a stack machine

- no registers
- instead it has an *expression stack* (onto which values are loaded)



word array (1 word = 4 bytes)  
need not be big (e.g. 32 words » 32 registers)

esp ... expression stack pointer

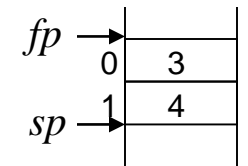


# How a Stack Machine Works

## Example

statement  $i = i + j * 5;$

assume the following values of  $i$  and  $j$



## Simulation

*instructions*    *stack*

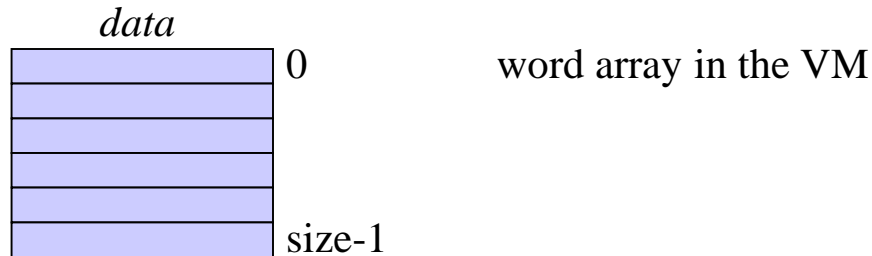
load0	<table border="1"><tr><td>3</td></tr></table>	3	load variable from address 0 (i.e. $i$ )		
3					
load1	<table border="1"><tr><td>3</td><td>4</td></tr></table>	3	4	load variable from address 1 (i.e. $j$ )	
3	4				
const5	<table border="1"><tr><td>3</td><td>4</td><td>5</td></tr></table>	3	4	5	load constant 5
3	4	5			
mul	<table border="1"><tr><td>3</td><td>20</td></tr></table>	3	20	multiply the two topmost stack elements	
3	20				
add	<table border="1"><tr><td>23</td></tr></table>	23	add the two topmost stack elements		
23					
store0		store the topmost stack element to address 0			

At the end of every statement the expression stack is empty!

# Data Areas of the *m*VM



## Global variables

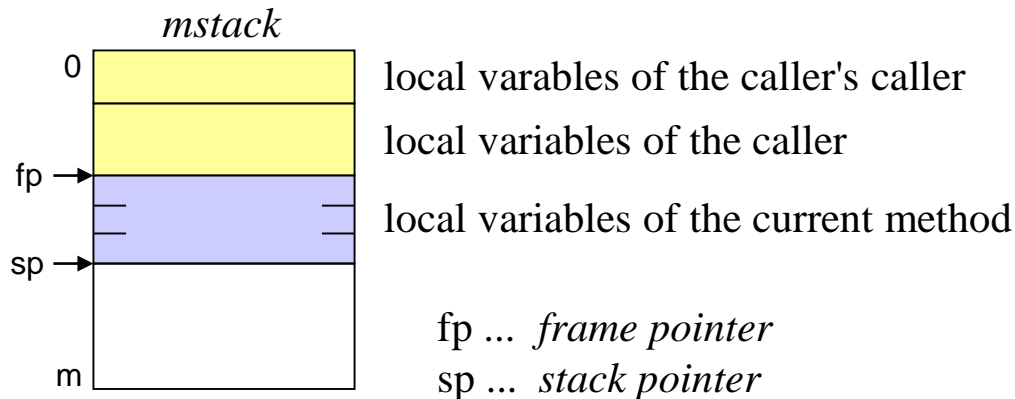
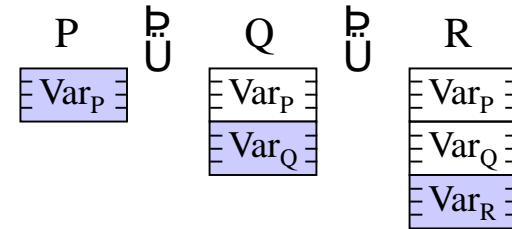


- area of fixed size
- global variables live during the whole program
- every variable occupies 1 word (4 bytes)
- global variables are addressed by word numbers  
e.g. *getstatic 2* loads the variable at address 2 from *data* to *estack*

# Data Areas of the *mJVM*

## Local variables

- are allocated in a *stack frame*
- every method invocation has its own stack frame
- frames are managed in a stack-like way



- local variables are addressed relative to *fp*
- every variable occupies 1 word (4 bytes)
- local variables are addressed by word numbers  
e.g. *load0* loads the variable at offset 0 from *fp* to *estack*

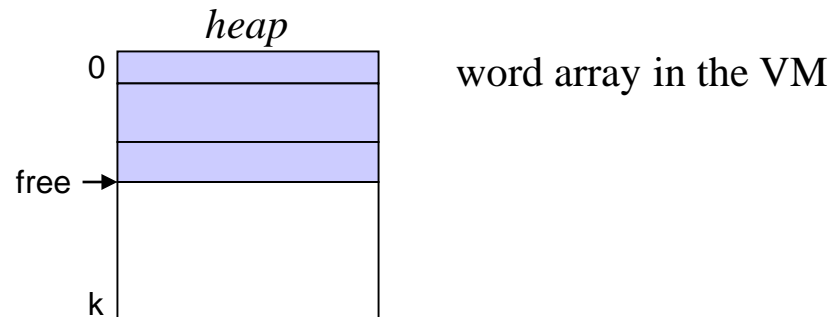


# Data Areas of the *mJVM*



## Heap

- contains class objects and array objects

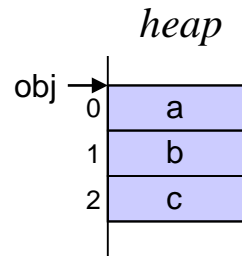


- New objects are allocated at the position *free* (and *free* is incremented); this is done by the VM instructions *new* and *newarray*
- Objects are never deallocated in MicroJava (no garbage collector)
- Pointers are word addresses relative to the beginning of the heap

# Data Areas of the JVM

## class objects

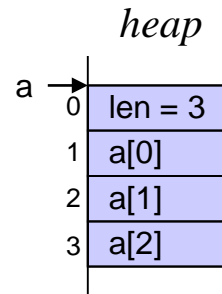
```
class T {
    int a, b;
    char c;
}
T obj = new T;
```



- every field occupies 1 word (4 bytes)
- addressed by word numbers relative to *obj*

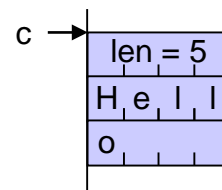
## array objects

```
int[] a;
a = new int[3];
```



- array length is stored in the array object
- every element occupies 1 word (4 bytes)

```
char[] c = new char[5];
```

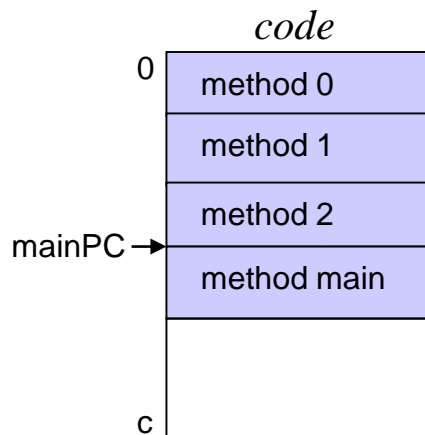


- *char* arrays are byte arrays
- but their length is a multiple of 4 bytes

# Code Area of the *m*VM

## Code

- byte array of fixed size
- methods are allocated consecutively
- *mainPC* points to the *main()* method



byte array in the VM

### *special registers of the VM*

fp frame pointer  
 sp stack pointer (mstack)  
 esp stack pointer (estack)  
 pc program counter



# Instruction Set of the mJVM

## Bytecodes (similar to Java bytecodes)

- very compact: most instructions are just 1 byte long
- untyped (the Java VM encodes operand types in instructions)

*MicroJava*

load0  
load1  
add

*Java*

iload0	fload0
iload1	fload1
iadd	fadd

reason: the Java bytecode verifier can use the operand types to check the integrity of the program

## Instruction format

very simple compared to Intel, PowerPC or SPARC

Code = {Instruction}.  
Instruction = opcode {operand}.

opcode ... 1 byte  
operand ... 1, 2 or 4 bytes

## Examples

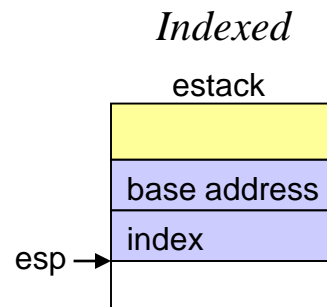
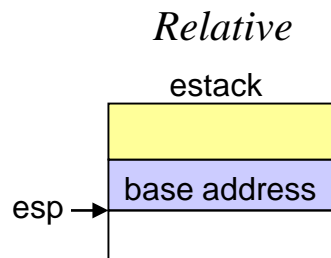
0 operands	add	has 2 implicit operands on the stack
1 operand	load 7	
2 operands	enter 0, 2	method entry

# Instruction Set of the *mJVM*

## Addressing modes

How can operands be accessed?

<i>addressing mode</i>	<i>example</i>	
• <b>Immediate</b>	const 7	for constants
• <b>Local</b>	load 3	for local variables on <i>mstack</i>
• <b>Static</b>	getstatic 3	for global variables in <i>data</i>
• <b>Stack</b>	add	for loaded values on <i>estack</i>
• <b>Relative</b>	getfield 3	for object fields (load $heap[pop() + 3]$ )
• <b>Indexed</b>	aload	for array elements (load $heap[pop() + 1 + pop()]$ )



# Instruction Set of the mJVM



## Load/store of local variables

<b>load</b>	b	... ..., val	<u>Load</u> push(local[b]);
<b>load&lt;n&gt;</b>		... ..., val	<u>Load</u> (n = 0..3) push(local[n]);
<b>store</b>	b	..., val ...	<u>Store</u> local[b] = pop();
<b>store&lt;n&gt;</b>		..., val ...	<u>Store</u> (n = 0..3) local[n] = pop();

*operand lengths*

b ... byte

s ... short (2 bytes)

w ... word (4 bytes)

## Load/store of global variables

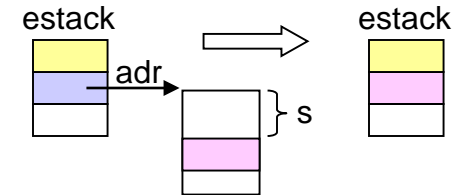
<b>getstatic</b>	s	... ..., val	<u>Load static variable</u> push(data[s]);
<b>putstatic</b>	s	..., val ...	<u>Store static variable</u> data[s] = pop();

# Instruction Set of the mJVM



## Load/store of object fields

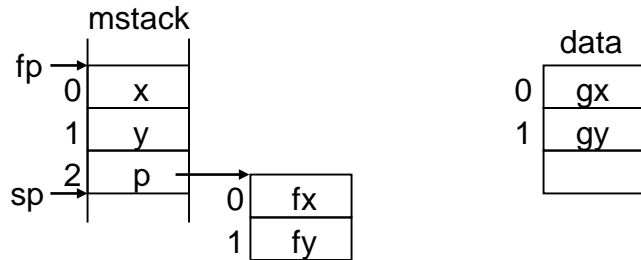
<b>getfield</b>	s	..., adr ..., val	<u>Load object field</u> adr = pop(); push(heap[adr+s]);
<b>putfield</b>	s	..., adr, val ...	<u>Store object field</u> val = pop(); adr = pop(); heap[adr+s] = val;



## Loading constants

<b>const</b>	w	... ..., val	<u>Load constant</u> push(w);
<b>const&lt;n&gt;</b>		... ..., val	<u>Load constant</u> (n = 0..5) push(n);
<b>const_m1</b>		... ..., val	<u>Load minus one</u> push(-1);

# Examples: Loading and Storing



	<i>code</i>	<i>bytes</i>	<i>stack</i>
<b>x = y;</b>	load1 store0	1 1	y -
<b>gx = gy;</b>	getstatic 1 putstatic 0	3 3	gy -
<b>p.fx = p.fy;</b>	load2 load2 getfield 1 putfield 0	1 1 3 3	p p p p p.fy -



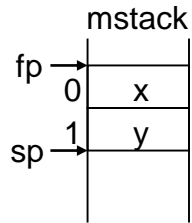
# Instruction Set of the *mJVM*

## Arithmetic

<b>add</b>	..., val1, val2 ..., val1+val2	<u>Add</u> push(pop() + pop());
<b>sub</b>	..., val1, val2 ..., val1-val2	<u>Subtract</u> push(-pop() + pop());
<b>mul</b>	..., val1, val2 ..., val1*val2	<u>Multiply</u> push(pop() * pop());
<b>div</b>	..., val1, val2 ..., val1/val2	<u>Divide</u> y = pop(); push(pop() / y);
<b>rem</b>	..., val1, val2 ..., val1%val2	<u>Remainder</u> y = pop(); push(pop() % y);
<b>neg</b>	..., val ..., -val	<u>Negate</u> push(-pop());
<b>shl</b>	..., val, x ..., val1	<u>Shift left</u> x = pop(); push(pop() << x);
<b>shr</b>	..., val, x ..., val1	<u>Shift right</u> x = pop(); push(pop() >> x);



# Examples: Arithmetic Operations



	<i>code</i>	<i>bytes</i>	<i>stack</i>
<b>x + y * 3</b>	load0	1	x
	load1	1	x y
	const3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3

# Instruction Set of the *mJVM*

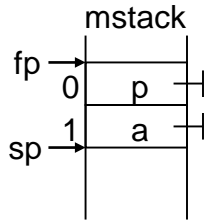


## Object creation

<b>new</b> s	... ..., adr	<u>New object</u> allocate area of s words; initialize area to all 0; push(adr(area));
<b>newarray</b> b	..., n ..., adr	<u>New array</u> n = pop(); if (b == 0) allocate byte array with n elements (+ length word); else if (b == 1) allocate word array with n elements (+ length word); initialize array to all 0; store n as the first word of the array; push(adr(array));



# Examples: Object Creation



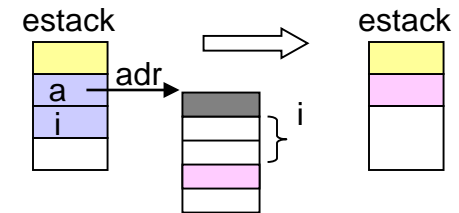
	<i>code</i>	<i>bytes</i>	<i>stack</i>	
<b>Person p = new Person;</b>	new 4	3	p	// assume: size(Person) = 4 words
	store0	1	-	
<b>int[] a = new int[5];</b>	const5	1	5	
	newarray 1	2	a	
	store1	1	-	

# Instruction Set of the *mJVM*

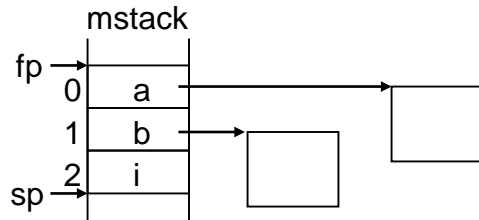


## Array access

<b>aload</b>	..., adr, i ..., val	<u>Load array element</u> i = pop(); adr = pop(); push(heap[adr+1+i]);
<b>astore</b>	...,adr, i, val ...	<u>Store array element</u> val = pop(); i = pop(); adr = pop(); heap[adr+1+i] = val;
<b>baload</b>	..., adr, i ..., val	<u>Load byte array element</u> i = pop(); adr = pop(); x = heap[adr+1+i/4]; push(byte i%4 of x);
<b>bastore</b>	...,adr, i, val ...	<u>Store byte array element</u> val = pop(); i = pop(); adr = pop(); x = heap[adr+1+i/4]; set byte i%4 in x to val; heap[adr+1+i/4] = x;
<b>arraylength</b>	..., adr ..., len	<u>Get array length</u> adr = pop(); push(heap[adr]);



# Example: Array Access



	<i>code</i>	<i>bytes</i>	<i>stack</i>
<b>a[i] = b[i+1];</b>	load0	1	a
	load2	1	a i
	load1	1	a i b
	load2	1	a i b i
	const1	1	a i b i 1
	add	1	a i b i+1
	aload	1	a i b[i+1]
	astore	1	-

# Instruction Set of the mJVM

## Stack manipulation

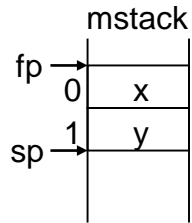
<b>pop</b>	..., val ...	<u>Remove topmost stack element</u> dummy = pop();
------------	-----------------	---

## Jumps

<b>jmp</b> s	... ...	<u>Jump unconditionally</u> pc = s;
<b>j&lt;cond&gt;</b> s	..., x, y ...	<u>Jump conditionally</u> (eq,ne,lt,le,gt,ge) y = pop(); x = pop(); if (x cond y) pc = s;



# Example: Jumps



	<i>code</i>	<i>bytes</i>	<i>stack</i>
if (x > y) ...	load0	1	x
	load1	1	x y
	jle ...	3	-

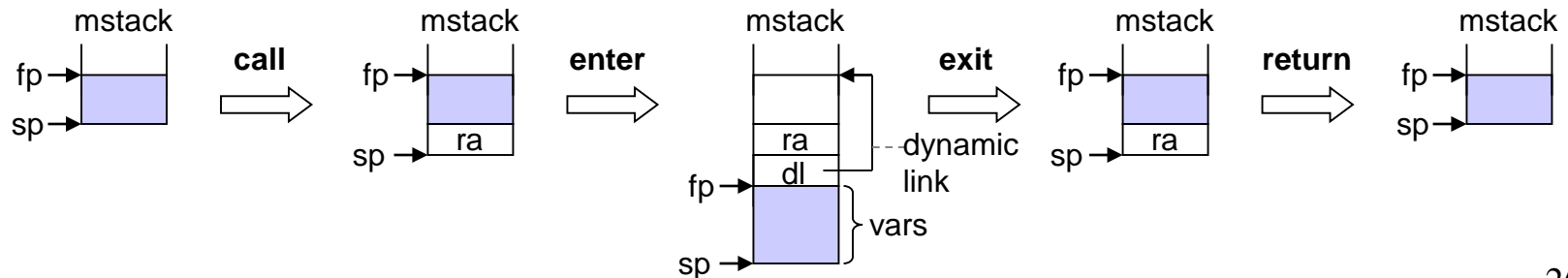


# Instruction Set of the *mJVM*

## Method call

<b>call</b>	s	...	...	<u>Call method</u> PUSH(pc+3); pc = s;
<b>enter</b>	b1, b2	...	...	<u>Enter method</u> pars = b1; vars = b2; // in words PUSH(fp); fp = sp; sp = sp + vars; initialize frame to 0; for (i=pars-1; i>=0; i--) local[i] = pop();
<b>exit</b>		...	...	<u>Exit method</u> sp = fp; fp = POP();
<b>return</b>		...	...	<u>Return</u> pc = POP();

PUSH and POP work on *mstack*





# Instruction Set of the mJVM

## Input/output

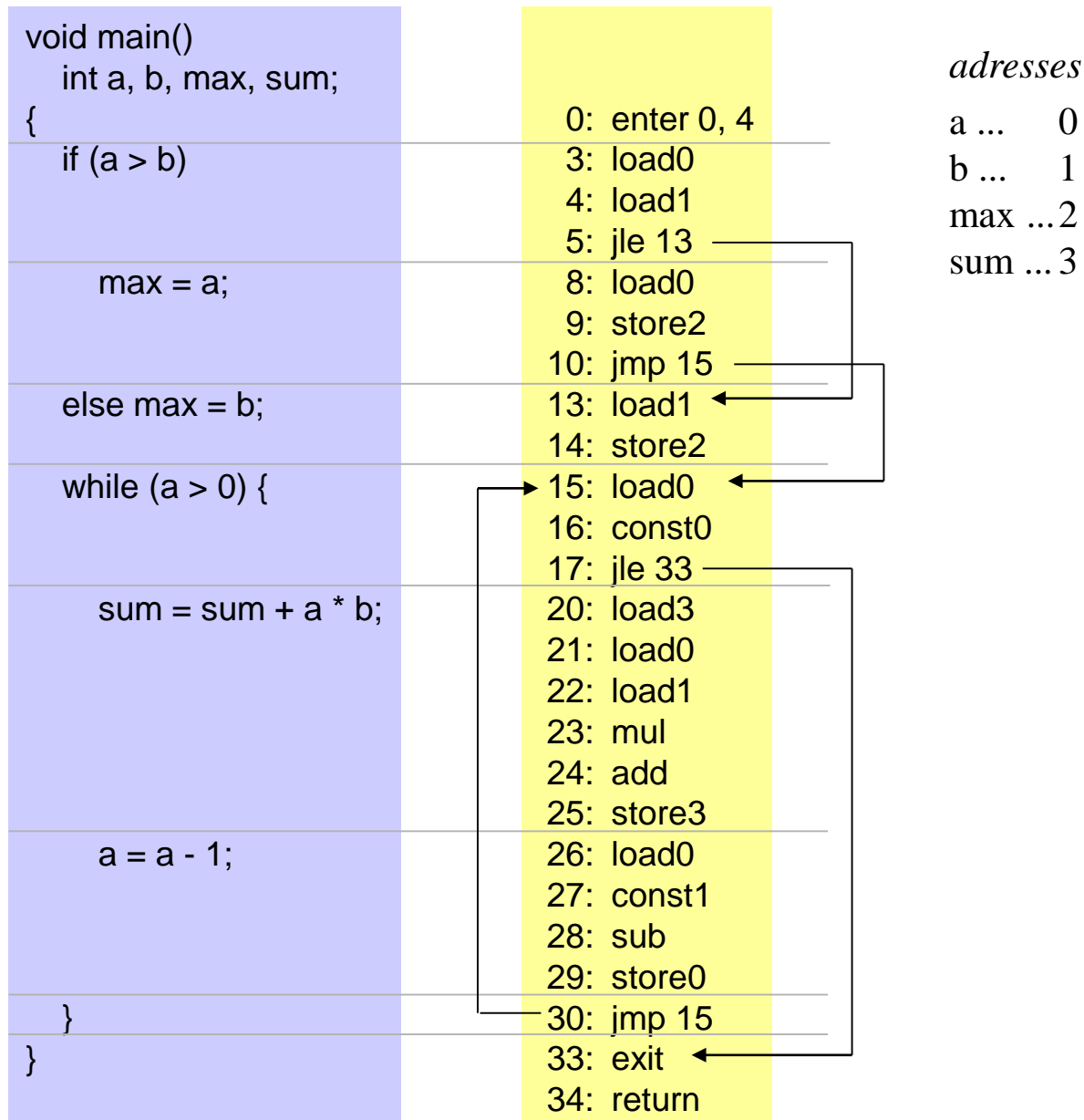
<b>read</b>	... ..., val	<u>Read</u> x = readInt(); push(x);
<b>print</b>	..., val, width ...	<u>Print</u> w = pop(); writeInt(pop(), w);
<b>bread</b>	... ..., val	<u>Read byte</u> ch = readChar(); push(ch);
<b>bprint</b>	..., val, width ...	<u>Print</u> w = pop(); writeChar(pop(), w);

input from System.in  
output to System.out

## Miscellaneous

<b>trap</b>	b	... ...	<u>Throw exception</u> print error message b; stop execution;
-------------	---	------------	---

# Example



## 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

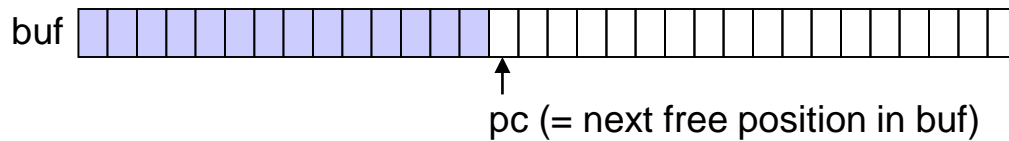
6.9 Methods



# Code Buffer

## Data structure

byte array in memory, because some instructions have to be patched later.



## Emitting instructions

simple, because MicroJava has a simple instruction format

```
class Code {
    private static byte[] buf = new
byte[3000];
    public static int pc = 0;

    public static void put (int x); {
        buf[pc++] = (byte)x;
    }
    public static void put2 (int x) {
        put(x >> 8); put(x);
    }
    public static void put4 (int x) {
        put2(x >> 16); put2(x);
    }
    ...
}
```

instruction codes are declared in class *Code*

```
static final int
load      = 1,
load0    = 2,
load1    = 3,
load2    = 4,
load3    = 5,
store    = 6,
store0   = 7,
store1   = 8,
store2   = 9,
store3   = 10,
getstatic = 11,
... ;
```

e.g., emitting *load 7*

```
Code.put(Code.load);
Code.put(7);
```

e.g.: emitting *load2*

```
Code.put(Code.load0 + 2);
```

## 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

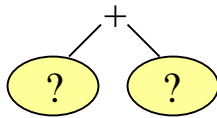
6.8 Control Structures

6.9 Methods

# Operands During Code Generation

## Example

we want to add two values



desired code pattern

```
load operand 1
load operand 2
add
```

**Depending on the operand kind we must generate different load instructions**

<i>operand kind</i>	<i>instruction to be generated</i>
• constant	const val
• local variable	load a
• global variable	getstatic a
• object field	getfield a
• array element	aload
• loaded value on the stack	---

We need a descriptor, which gives us all the necessary information about operands

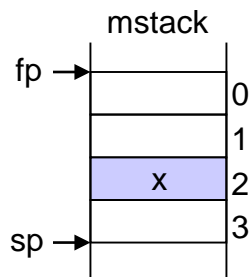
# Operand Descriptors



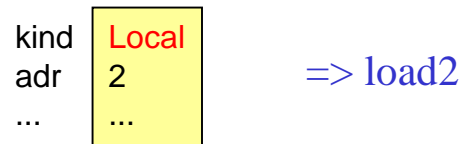
Descriptors holding information about variables, constants and expressions

## Example

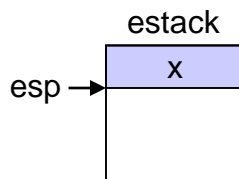
Local variable  $x$  in a stack frame



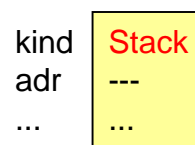
*described by the following operand descriptor*



After loading the value with *load2* it is on *estack* now



*described by the following operand descriptor*



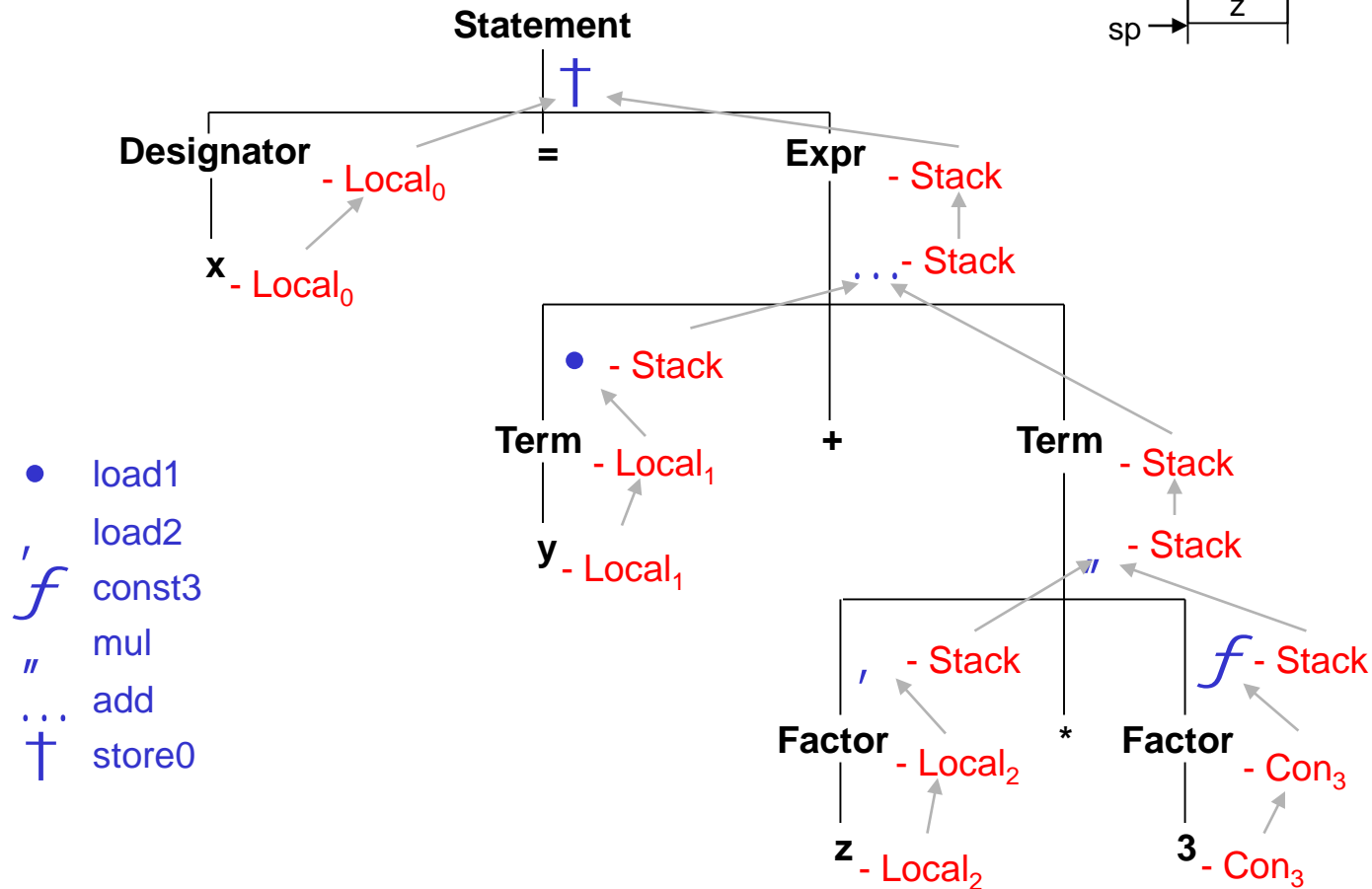
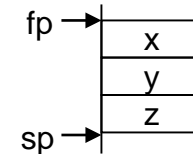


# Example: Processing of Operands



Most parsing methods return operands (as a result of their translation process)

Example: translating the assignment `x = y + z * 3;`



# Operand Kinds



<i>operand kind</i>	<i>operand code</i>	<i>info about operands</i>	
constant	<b>Con</b> = 0	constant value	
local variable	<b>Local</b> = 1	address	<p>A diagram showing a stack structure labeled 'mstack'. It consists of three rectangular boxes stacked vertically. The top two boxes are white, and the bottom one is blue. An arrow labeled 'fp' points to the top of the stack. A bracket on the right side of the top two boxes is labeled 'adr'.</p>
global variable	<b>Static</b> = 2	address	<p>A diagram showing a stack structure labeled 'data'. It consists of three rectangular boxes stacked vertically. The top two boxes are white, and the bottom one is blue. A bracket on the right side of the top two boxes is labeled 'adr'.</p>
value on the stack	<b>Stack</b> = 3	---	<p>A diagram showing a stack structure labeled 'estack'. It consists of three rectangular boxes stacked vertically. The middle box is blue. An arrow labeled 'esp' points to the bottom of the stack.</p>
object field	<b>Fld</b> = 4	offset	<p>A diagram showing a stack structure labeled 'estack'. It consists of three rectangular boxes stacked vertically. The middle box is pink and labeled 'adr'. An arrow labeled 'esp' points to the bottom of the stack. An arrow points from the 'adr' box to a separate stack structure. This second stack has three boxes, with the bottom one blue. A bracket on the right side of the top two boxes of this second stack is labeled 'offset'.</p>
array element	<b>Elem</b> = 5	---	<p>A diagram showing a stack structure labeled 'estack'. It consists of three rectangular boxes stacked vertically. The middle box is pink and labeled 'adr', and the bottom box is pink and labeled 'idx'. An arrow labeled 'esp' points to the bottom of the stack. An arrow points from the 'adr' box to a separate stack structure. This second stack has three boxes, with the bottom one blue. A bracket on the right side of the top two boxes of this second stack is labeled 'idx'.</p>
method	<b>Meth</b> = 6	address, method obj.	

# Class Operand



```
class Operand {
    static final int Con = 0, Local = 1, Static = 2, Stack = 3, Fld = 4, Elem = 5, Meth = 6;

    int    kind;    // Con, Local, Static, ...
    Struct type;    // type of the operand
    int    val;     // Con: constant value
    int    adr;     // Local, Static, Fld, Meth: address
    Obj    obj;     // Meth: method object
}
```

## Constructors for creating operands

```
public Operand (Obj obj) {
    type = obj.type; val = obj.val; adr = obj.adr;
    switch (obj.kind) {
        case Obj.Con:    kind = Con; break;
        case Obj.Var:    if (obj.level == 0) kind = Static; else kind = Local;
                        break;
        case Obj.Meth:   kind = Meth; this.obj = obj; break;
        default:         error("cannot create operand");
    }
}
```

creates an operand from  
a symbol table object

```
public Operand (int val) {
    kind = Con; type = Tab.intType; this.val = val;
}
```

creates an operand from  
a constant value



# Loading Values

**given:** a value described by an operand descriptor (Con, Local, Static, ...)

**wanted:** code that loads the value onto the expression stack

```
public static void load (Operand x) { // method of class Code
  switch (x.kind) {
    case Operand.Con:
      if (0 <= x.val && x.val <= 5) put(const0 + x.val);
      else if (x.val == -1) put(const_m1);
      else { put(const_); put4(x.val); }
      break;
    case Operand.Static:
      put(getstatic); put2(x.adr); break;
    case Operand.Local:
      if (0 <= x.adr && x.adr <= 3) put(load0 + x.adr);
      else { put(load); put(x.adr); }
      break;
    case Operand.Fld: // assert: object base address is on the stack
      put(getfield); put2(x.adr); break;
    case Operand.Elem: // assert: base address and index are on stack
      if (x.type == Tab.charType) put(baload); else put(aload);
      break;
    case Operand.Stack: break; // nothing (already loaded)
    default: error("cannot load this value");
  }
  x.kind = Operand.Stack;
}
```

## Case analysis

depending on the operand kind we have to generate different load instructions

resulting operand is always a *Stack* operand

# Example: Loading Variables

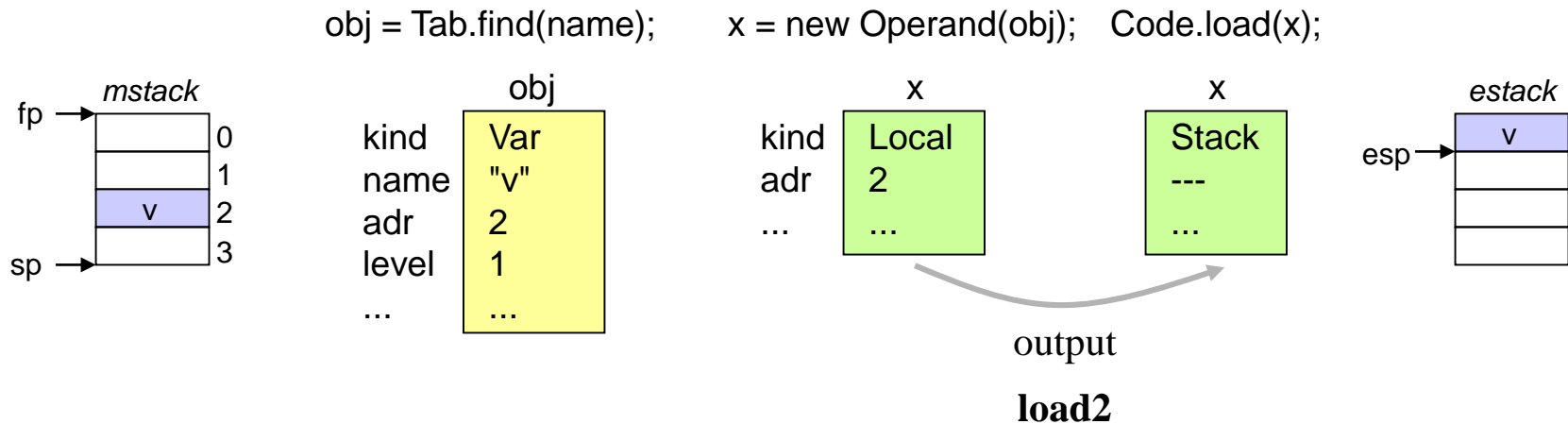
## Description by an ATG

```

Factor <- x>      (. String name; .)
= ident <- name> (. Obj obj = Tab.find(name);      // obj.kind = Var | Con
                  Operand x = new Operand(obj);    // x.kind = Local | Static | Con
                  Code.load(x);                    // x.kind = Stack
                  .) .

```

## Visualisation



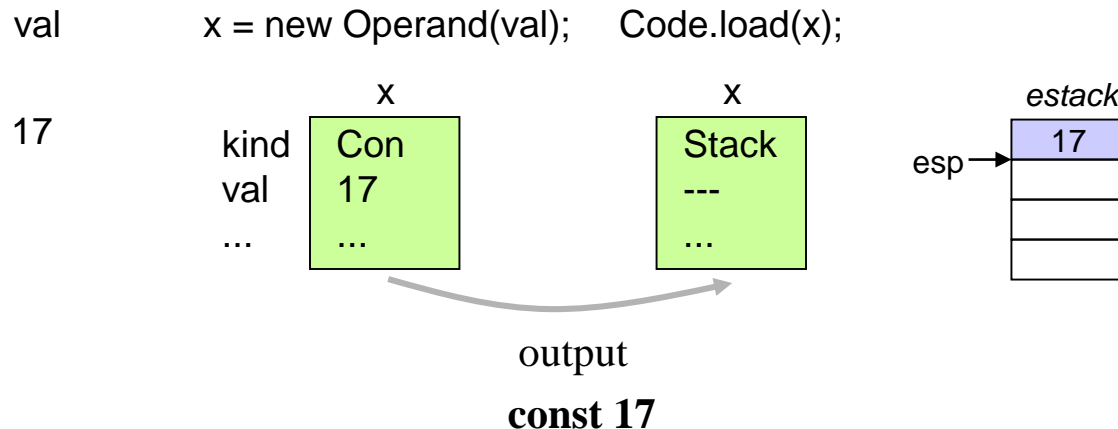
# Example: Loading Constants

## Description by an ATG

```

Factor <- x>      (. int val; .)
= number <- val>  (. Operand x = new Operand(val); // x.kind = Con
                  Code.load(x);                  // x.kind = Stack
                  .)
  
```

## Visualisation



# Loading Object Fields

var.f

**Context conditions** (make sure that your compiler checks them)

Designator<sub>0</sub> = Designator<sub>1</sub> "." ident .

- The type of *Designator*<sub>1</sub> must be a class.
- *ident* must be a field of *Designator*<sub>1</sub>.

## Description by an ATG

```

Designator <- x>      (. String name, fName; .)
= ident <- name>      (. Obj obj = Tab.find(name);
                      Operand x = new Operand(obj); .)
  { "." ident <- fName> (. if (x.type.kind == Struct.Class) {
                        Code.load(x);
                        Obj fld = Tab.findField(fName, x.type);
                        x.kind = Operand.Fld;
                        x.adr = fld.adr;
                        x.type = fld.type;
                        } else error(name + " is not an object"); .)

  | ...
  }.

```

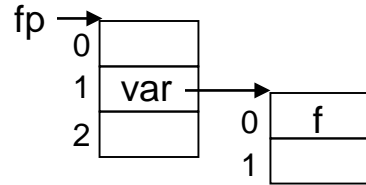
looks up *fName* in the field list of *x.type*

creates a *Fld* operand

# Operand Sequence



var.f

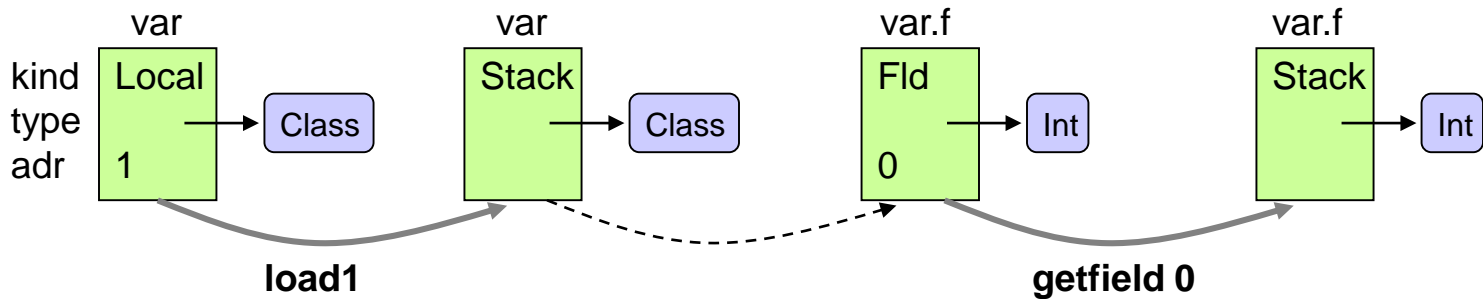


```

Designator <- x>      (. String name, fName; .)
= ident <- name>        (. Obj obj = Tab.find(name);
                        Operand x = new Operand(obj); .)

{ "." ident <- fName>   (. if (x.type.kind == Struct.Class) {
                        Code.load(x);
                        obj = Tab.findField(fName, x.type);
                        x.kind = Operand.Fld;
                        x.adr = obj.adr;
                        x.type = obj.type;
                        } else error(name + " is not an object"); .)

| ...
}
    
```





# Loading Array Elements

a[i]

## Context conditions

Designator<sub>0</sub> = Designator<sub>1</sub> "[" Expr "]" .

- The type of *Designator*<sub>1</sub> must be an array.
- The type of *Expr* must be *int*.

## Description by an ATG

```

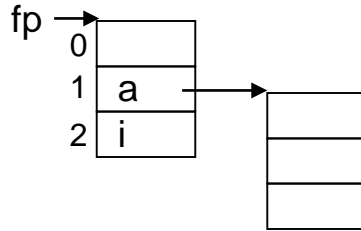
Designator <- x>      (. String name; Operand x, y; .)
= ident <- name>      (. Obj obj = Tab.find(name); x = new Operand(obj); .)
{ ...
| "["                (. Code.load(x); .)
  Expr <- y>         (. if (x.type.kind == Struct.Arr) {
                      if (y.type.kind != Struct.Int) error("index must be of type int");
                      Code.load(y);
                      x.kind = Operand.Elem; ← creates an Elem operand
                      x.type = x.type.elemType;
                      } else error(name + " is not an array"); .)
  "]"
}.

```

# Operand Sequence



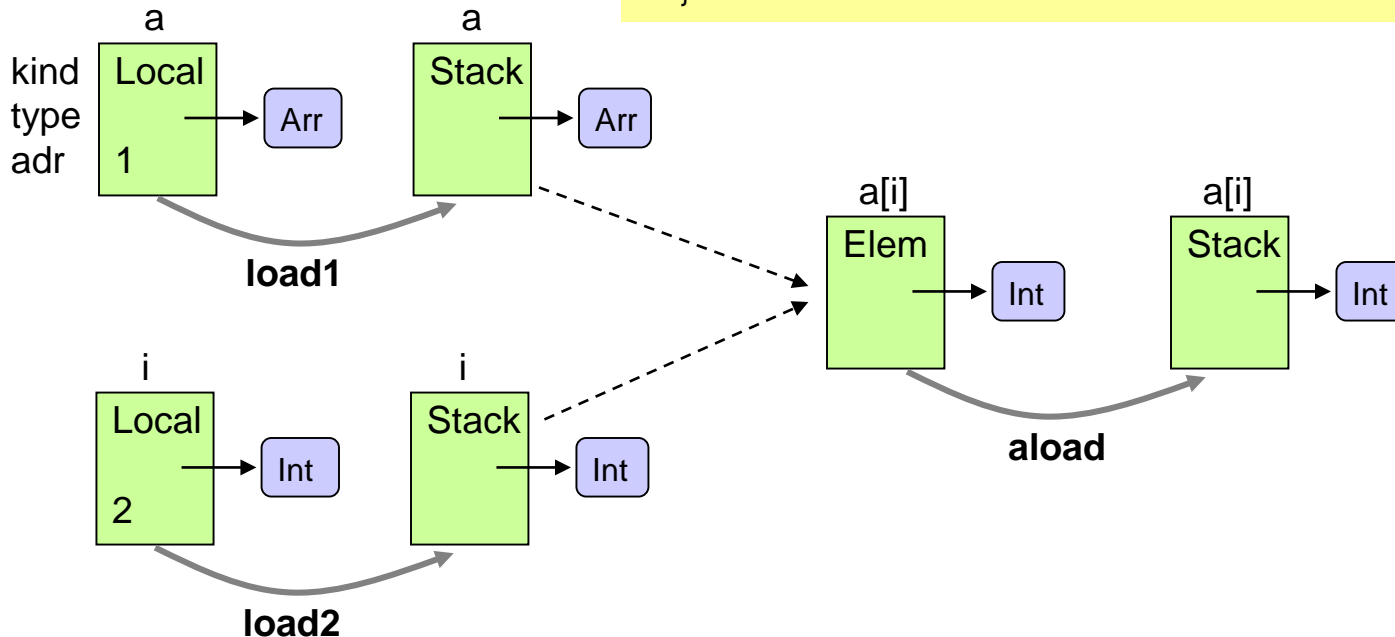
a[i]



```

Designator <- x>      (. String name; Operand x, y; .)
= ident <- name>      (. Obj obj = Tab.find(name); x = new Operand(obj); .)
{ ...
| "["                (. Code.load(x); .)
  Expr <- y>         (. if (x.type.kind == Struct.Arr) {
                      if (y.type.kind != Struct.Int)
                        error("index must be of type int");
                      Code.load(y);
                      x.kind = Operand.Elem;
                      x.type = x.type.elemType;
                    } else error(name + " is not an array"); .)

  "]"
}.
    
```



## 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

# Compiling Expressions



**Scheme** for  $x + y + z$

```
load x
load y
add
load z
add
```

**Context conditions**

Expr = "-" Term.

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

Expr<sub>0</sub> = Expr<sub>1</sub> Addop Term.

- *Expr<sub>1</sub>* and *Term* must be of type *int*.

**Description by an ATG**

```
Expr <- x>      (. Operand x, y; int op; .)
= ( Term <- x>
  | "-" Term <- x>  (. if (x.type != Tab.intType) error("operand must be of type int");
                    if (x.kind == Operand.Con) x.val = -x.val;
                    else {
                      Code.load(x); Code.put(Code.neg);
                    } .)
)
{ ("+"            (. op = Code.add; .)
  | "-"          (. op = Code.sub; .)
)
  Term <- y>      (. Code.load(x); .)
                (. Code.load(y);
                  if (x.type != Tab.intType || y.type != Tab.intType)
                    error("operands must be of type int");
                  Code.put(op); .)
}.
```



# Compiling Terms

Term<sub>0</sub> = Term<sub>1</sub> Mulop Factor.

- Term<sub>1</sub> and Factor must be of type int.

```
Term <- x>          (. Operand x, y; int op; .)
= Factor <- x>
  { ( "*"          (. op = Code.mul; .)
    | "/"          (. op = Code.div; .)
    | "%"          (. op = Code.rem; .)
    )              (. Code.load(x); .)
  Factor <- y>      (. Code.load(y);
                    if (x.type != Tab.intType || y.type != Tab.intType)
                      error("operands must be of type int");
                    Code.put(op); .)
  }.
}
```

# Compiling Factors



Factor = "new" ident.

- *ident* must denote a class.

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

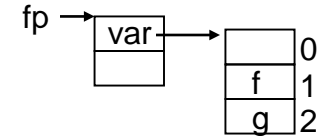
- *ident* must denote a type.
- The type of *Expr* must be *int*.

```
Factor <- x>      (. Operand x; int val; String name; .)
= Designator <- x> // function calls see later
| number <- val>   (. x = new Operand(val); .)
| charCon <- val> (. x = new Operand(val); x.type = Tab.charType; .)
| "(" Expr <- x> ")"
| "new" ident <- name> (. Obj obj = Tab.find(name); Struct type = obj.type; .)
( "["
  Expr <- x> "]"
  (. if (obj.kind != Obj.Type) error("type expected"); .)
  (. if (x.type != Tab.intType) error("array size must be of type int");
    Code.load(x);
    Code.put(Code.newarray);
    if (type == Tab.charType) Code.put(0); else Code.put(1);
    type = new Struct(Struct.Arr, type); .)
|
  (. if (obj.kind != Obj.Type || type.kind != Struct.Class)
    error("class type expected");
    Code.put(Code.new_); Code.put2(type.nFields);
)
.
(. x = new Operand(); x.kind = Operand.Stack; x.type = type; .)
```

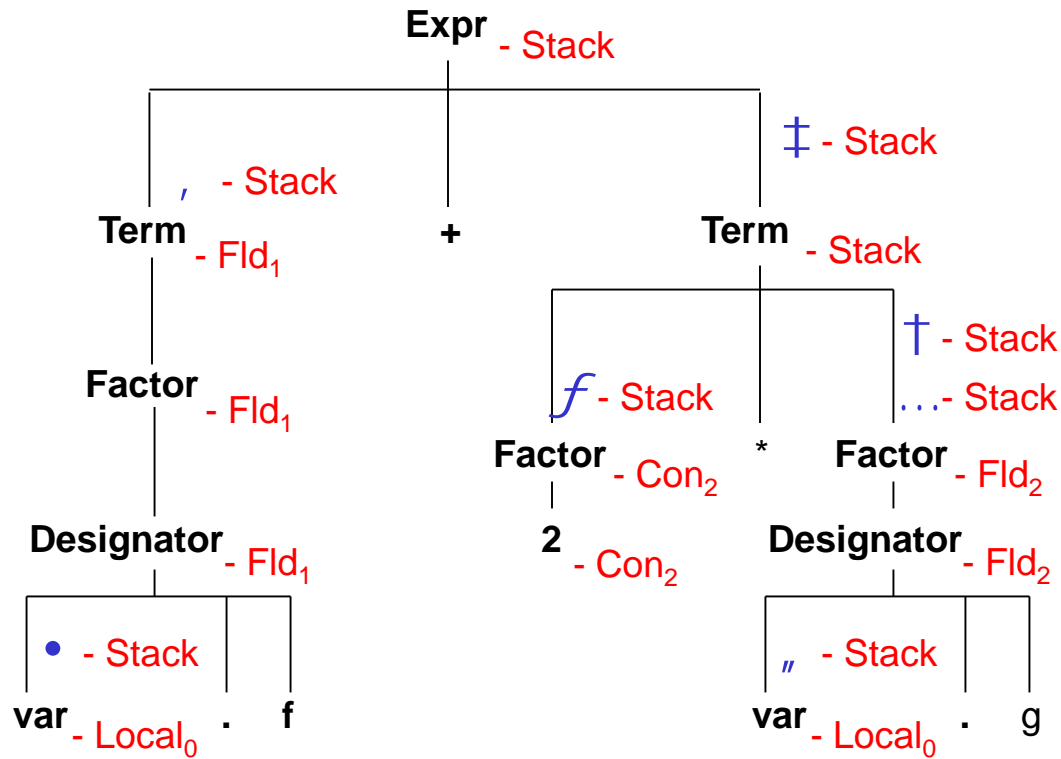
# Example



var.f + 2 \* var.g



- load0
- , getfield 1
- f* const2
- load0
- " getfield 2
- ... getfield 2
- † mul
- ‡ add



## 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



# Code Patterns for Assignments

```
designator = expr ;
```

4 cases depending on the kind of the designator on the left-hand side

<code>localVar = expr;</code>	<code>globalVar = expr;</code>	<code>obj.f = expr;</code>	<code>a[i] = expr;</code>
<pre>... load expr ... store localVar</pre>	<pre>... load expr ... putstatic globalVar</pre>	<pre>load obj ... load expr ... putfield f</pre>	<pre>load a load i ... load expr ... astore</pre>

the blue instructions are already generated by *Designator!*

# Compiling Assignments

## Context condition

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*.

## Description by an ATG

```

Assignment      (. Operand x, y; .)
= Designator <- x> // this call may already generate code
  "=" Expr <- y>   (. if (y.type.assignableTo(x.type))
                    Code.assign(x, y); // x: Local | Static | Fld | Elem
                    // assign must load y
                    else
                    error("incompatible types in assignment");
                    .)
"
```

## Assignment compatibility

*y* is assignment compatible with *x*

- if *x* and *y* have the same type ( $x.type == y.type$ ), or
- *x* and *y* are arrays with the same element type, or
- *x* has a reference type (class or array) and *y* is *null*

## 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

# Conditional and Unconditional Jumps



## Unconditional jumps

```
jmp address
```

## Conditional jumps

```
... load operand1 ...  
... load operand2 ...  
jeq address
```

if (operand1 == operand2) jmp address

jeq	jump on equal
jne	jump on not equal
jlt	jump on less than
jle	jump on less or equal
jgt	jump on greater than
jge	jump on greater or equal

```
static final int  
    eq = 0,  
    ne = 1,  
    lt = 2,  
    le = 3,  
    gt = 4,  
    ge = 5;
```

in class *Code*

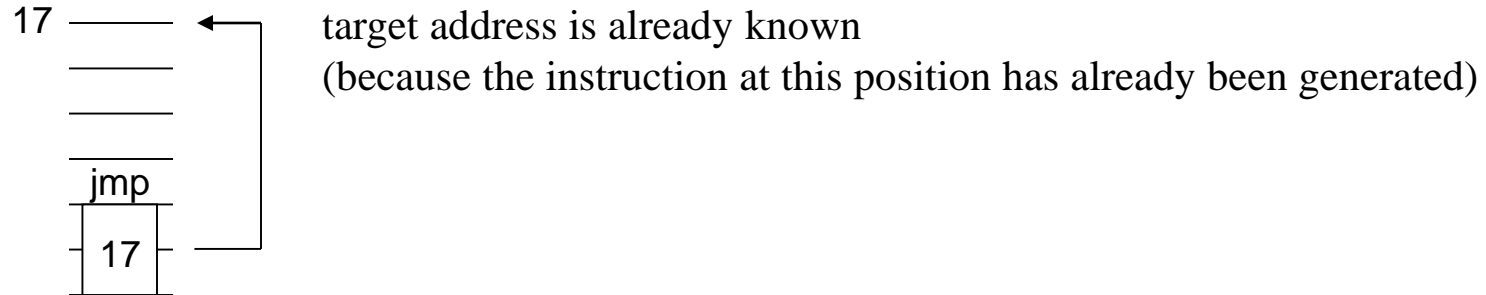
## Creation of jump instructions

```
Code.put(Code.jmp);  
Code.put2(address);
```

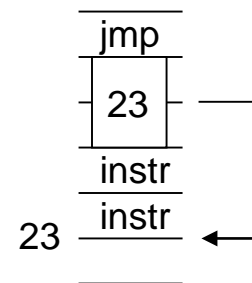
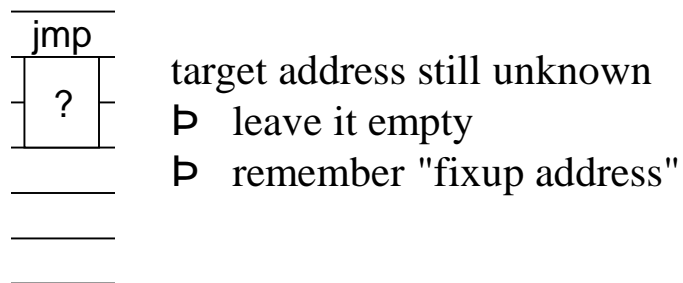
```
Code.put(Code.jeq + operator);  
Code.put2(address);
```

# Forward and Backward Jumps

## Backward jumps



## Forward jumps



patch it when the target address becomes known (fixup)

# Conditions

## Conditions

if ( a > b ) ...

⏟  
*Condition*

code pattern

load a

load b

jle ...

- Problem: the JVM has no compare instructions
  - ↳ *Condition* cannot generate a compare operation
- instead, *Condition* returns the compare operator; the comparison is then done in the jump instruction

```

Condition <- op>      (. int op; Operand x, y; .)
= Expr <- x>           (. Code.load(x); .)
  Relop <- op>
  Expr <- y>           (. Code.load(y);
                       if (!x.type.compatibleWith(y.type)) error("type mismatch");
                       if (x.type.isRefType() && op != Code.eq && op != Code.ne)
                           error("invalid compare"); .)

```



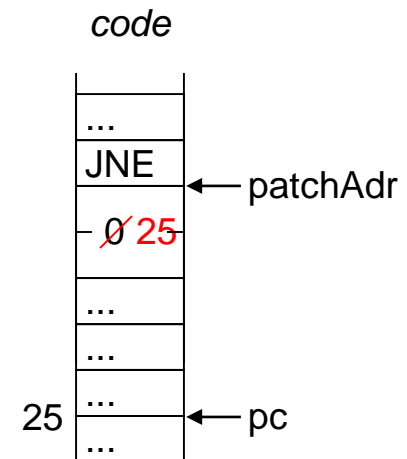
# Methods for Generating Jumps

```
class Code {
  private static final int
    eq = 0, ne = 1, lt = 2, le = 3, gt = 4, ge = 5;
  private static int[] inverse = {ne, eq, ge, gt, le, lt};
  ...
  // generate an unconditional jump to adr
  void putJump (int adr) {
    put(jmp); put2(adr);
  }

  // generate a conditional false jump (jump if not op)
  void putFalseJump (int op, int adr) {
    put(jeq + inverse[op]); put2(adr);
  }

  // patch the jump address at adr so that it leads to pc
  void fixup (int patchAdr) {
    put2(patchAdr, pc);
  }
}
```

new method of class Code



## 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**

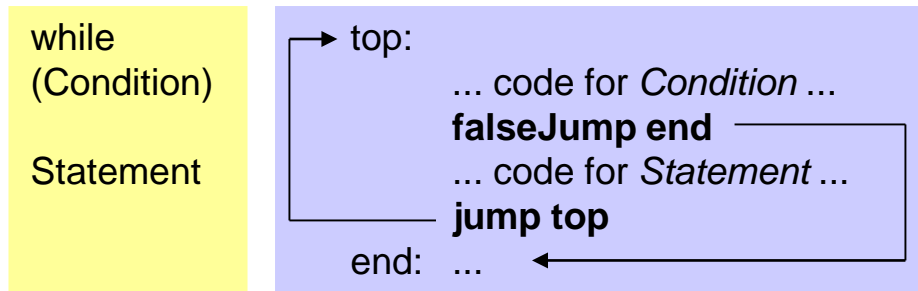
6.9 Methods





# while Statement

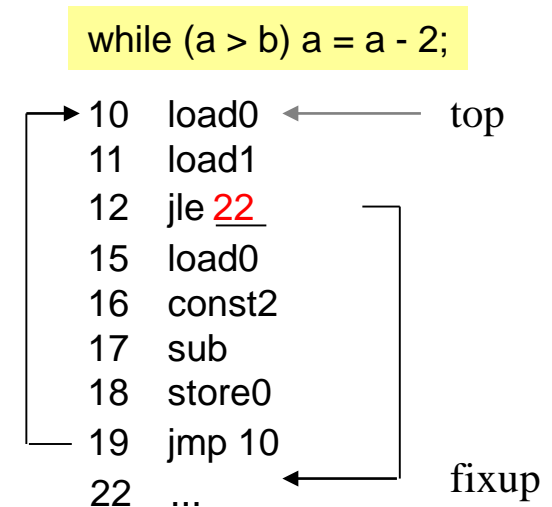
## Desired code pattern



## Description by an ATG

<b>WhileStatement</b>	(. int op; .)
= "while"	(. int top = Code.pc .)
"(" Condition <- op> ")"	(. Code.putFalseJump(op, 0); int adr = Code.pc - 2; .)
Statement	(. Code.putJump(top); Code.fixup(adr); .)
.	

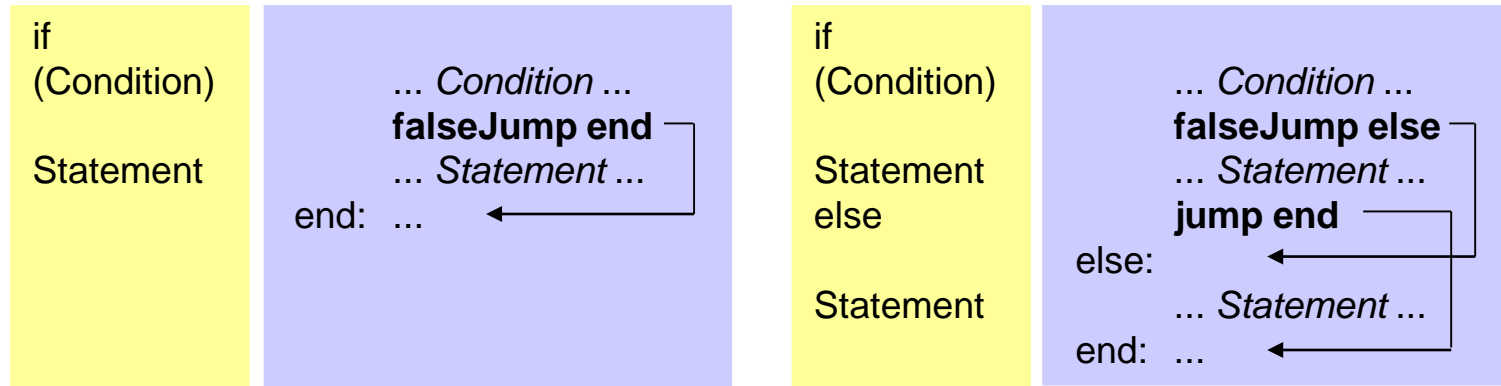
## Example



# if Statement



## Desired code pattern



## Description by an ATG

```

IfStatement          (. int op; .)
= "if"
  (" Condition <- op> ") (. Code.putFalseJump(op, 0);
                        int adr = Code.pc - 2; .)

  Statement
  ( "else"              (. Code.putJump(0);
                        int adr2 = Code.pc - 2;
                        Code.fixup(adr); .)

    Statement          (. Code.fixup(adr2); .)
    |                  (. Code.fixup(adr); .)
  ).
  
```

## Example

```

if (a > b) max = a; else max = b;

10 load0
11 load1
12 jle 20
15 load0
16 store2
17 jmp 22
20 load1
21 store2
22 ...
  
```

fixup(adr) points to line 20  
 fixup(adr2) points to line 22

## 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**

# Procedure Call

## Code pattern

m(a, b);	load a load b call m	parameters are passed on the <i>estack</i>
----------	----------------------------	--

## Description by an ATG

<b>Statement</b>	(. Operand x, y; ... .)
= Designator <- x>	(. Code.put(Code.call);
( ActPars <- x>	Code.put2(x.adr);
"=" Expr <- y> ";"	if (x.type != Tab.noType) Code.put(Code.pop); .)
... .	(. ... .)



# Function Call

## Code pattern

```

c = m(a, b);

```

load a      parameters are passed on the *estack*  
load b  
call m  
store c      function value is returned on the *estack*

## Standard functions

- ord('a')
- *ActPars* loads 'a' onto the *estack*
- the loaded value gets the type of *ordObj* (= *intType*) and *kind* = *Operand.Stack*

## Description by an ATG

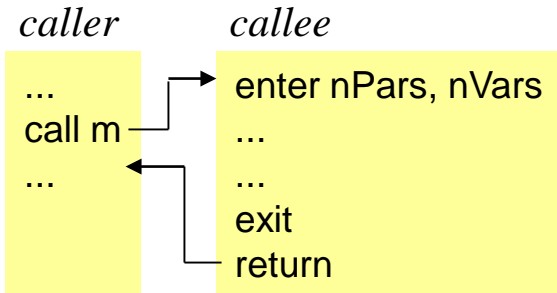
```

Factor <- x>      (. Operand x, m; .)
= Designator <- x>
  [ ActPars <- x>  (. if (x.type == Tab.noType) error("procedure called as a function");
                  if (x.obj == Tab.ordObj || x.obj == Tab.chrObj) ; // nothing
                  else if (x.obj == Tab.lenObj)
                    Code.put(Code.arraylength);
                  else {
                    Code.put(Code.call);
                    Code.put2(x.adr);
                  }
                  x.kind = Operand.Stack; .)

]
| ...

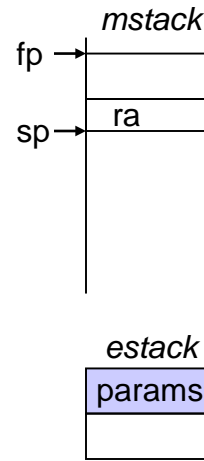
```

# Stack Frames



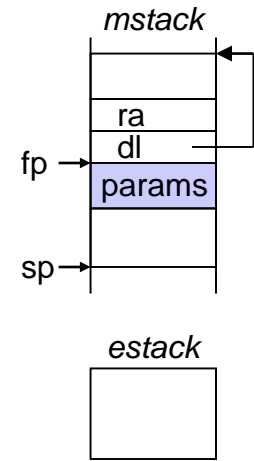
*enter* ... creates a stack frame  
*exit* ... removes a stack frame

## Method entry

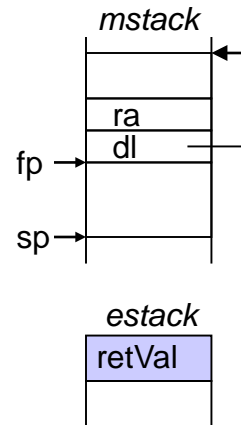


### enter nPars, nVars

```
PUSH(fp); // dynamic link
fp = sp;
sp = sp + nVars;
initialize frame to 0;
for (i=nPars-1; i>=0; i--)
    local[i] = pop();
```

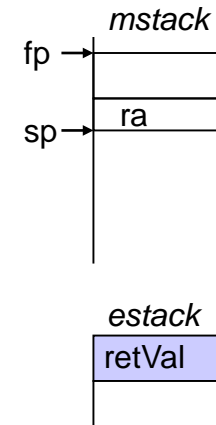


## Method exit



### exit

```
sp = fp;
fp = POP();
```



# Method Declaration



```
MethodDecl      (. Struct type; String name; int n; .)
= ( Type <- type>
  | "void"      (. type = Tab.noType; .)
  )
ident <- name>  (. curMethod = Tab.insert(Obj.Meth, name, type);
                Tab.openScope(); .)
"(" FormPars <- n> ")" (. curMethod.nPars = n;
                        if (name.equals("main")) {
                            Code.mainPc = Code.pc;
                            if (curMethod.type != Tab.noType) error("method main must be void");
                            if (curMethod.nPars != 0) error("main must not have parameters");
                        } .)

{ VarDecl }
"{"      (. curMethod.locals = Tab.curScope.locals;
          curMethod.adr = Code.pc;
          Code.put(Code.enter);
          Code.put(curMethod.nPars);
          Code.put(Tab.curScope.nVars); .)

{ Statement }
"}"      (. if (curMethod.type == Tab.noType) {
          Code.put(Code.exit); Code.put(Code.return_);
          } else { // end of function reached without a return statement
          Code.put(Code.trap); Code.put(1);
          }
          Tab.closeScope(); .)
.
```

# Formal Parameters

- are entered into the symbol table (as variables of the method scope)
- their number is counted

```
FormPars <- n>      (. int n = 0; .)
= [ FormPar         (. n++; .)
    { "," FormPar   (. n++; .)
    }
  ].
```

```
FormPar           (. Struct type; String name; .)
= Type <- type>
  ident <- name>   (. Tab.insert(Obj.Var, name, type); .)
.
```



# Actual Parameters

- load them to *estack*
- check if they are assignment compatible with the formal parameters
- check if the numbers of actual and formal parameters match

```

ActPars <̄ m>      (. Operand m, ap; .)
= "("              (. if (m.kind != Operand.Meth) { error("not a method"); m.obj = Tab.noObj; } .)
                   int aPars = 0;
                   int fPars = m.obj.nPars;
                   Obj fp = m.obj.locals; .)

  [ Expr <- ap>     (. Code.load(ap); aPars++;
                   if (fp != null) {
                       if (!ap.type.assignableTo(fp.type)) error("parameter type mismatch");
                       fp = fp.next;
                   } .)

  {" , " Expr <- ap> (. Code.load(ap); aPars++;
                   if (fp != null) {
                       if (!ap.type.assignableTo(fp.type)) error("parameter type mismatch");
                       fp = fp.next;
                   } .)

  }

  ]                (. if (aPars > fPars)
                   error("too many actual parameters");
                   else if (aPars < fPars)
                       error("too few actual parameters"); .)

  ")" .

```



# *return Statement*

## Statement

```
= ...  
| "return"  
  ( Expr <- x>      (. Code.load(x);  
                    if (curMethod.type == Tab.noType)  
                      error("void method must not return a value");  
                    else if (!x.type.assignableTo(curMethod.type))  
                      error("type of return value must match method type");  
                    .)  
  |                (. if (curMethod.type != Tab.noType) error("return value expected"); .)  
  )                (. Code.put(Code.exit);  
                    Code.put(Code.return_); .)  
  ", "  
  ; .
```



# Object File

## Contents of the object file in MicroJava

- information for the loader
  - code size (in bytes)
  - size of the global data area (in words)
  - address of the *main* method
- code

0	"MJ"
2	codeSize
6	dataSize
10	mainPc
14	code

The object file format in other languages is usually much more complex.