6. Code Generation

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

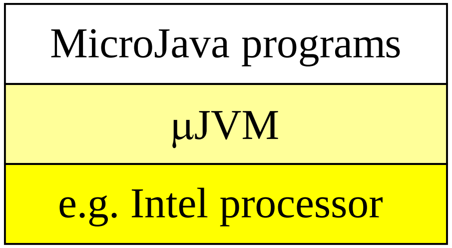
Output of the object file

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

Architecture of the MicroJava VM (μJVM)



MicroJava programs

μJVM

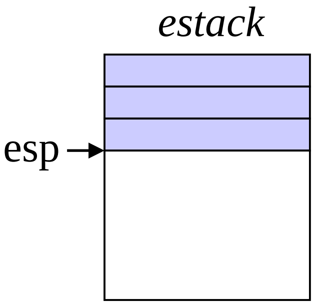
e.g. Intel processor

What is a virtual machine (VM)?

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

The μJVM 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)

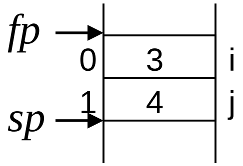
esp ... expression stack pointer

How a Stack Machine Works

Example

statement **i = i + j * 5;**

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



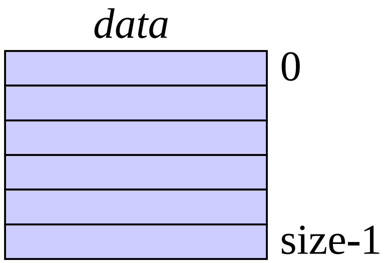
Simulation

instructions	*stack*	
load0	3	load variable from address 0 (i.e. *i*)
load1	3 4	load variable from address 1 (i.e. *j*)
const5	3 4 5	load constant 5
mul	3 20	multiply the two topmost stack elements
add	23	add the two topmost stack elements
store0		store the topmost stack element to address 0

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

Data Areas of the μJVM

Global variables



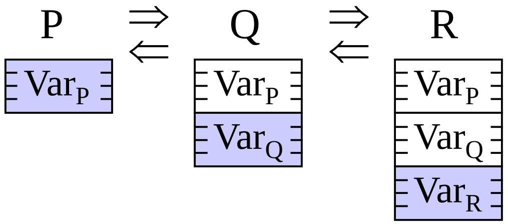
word array in the VM

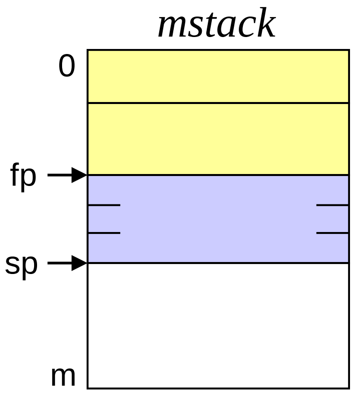
- area of fixed size
- global variables live during the whole program
- every variable occupies 1 word (4 bytes)
- global variables are addressed by word numbers
  e.g. *getstatic 2* loads the variable at address 2 from *data* to *estack*

Data Areas of the μJVM

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 varables of the caller's caller

local variables of the caller

local variables of the current method

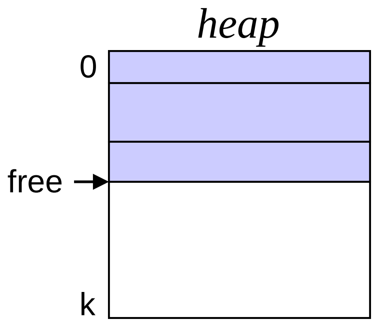
fp ... *frame pointer*
sp ... *stack pointer*

- 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 μJVM

Heap

- contains class objects and array objects



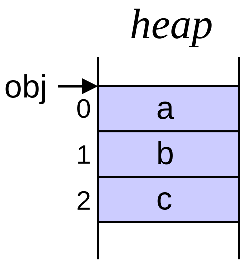
word array in the VM

- 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 μJVM

class objects

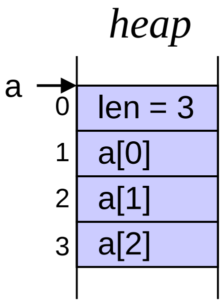
```
class T {
    int a, b;
    char c;
}
T obj = new T;
```



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

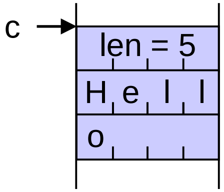
array objects

```
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];
```

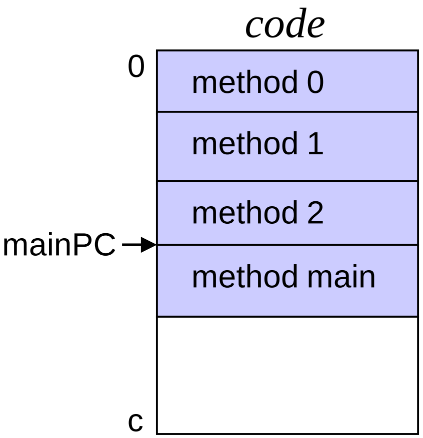


- *char* arrays are byte arrays
- but their length is a multiple of 4 bytes

Code Area of the μJVM

Code

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



code

0

method 0

method 1

method 2

mainPC →

method main

c

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 µJVM

Bytecodes (similar to Java bytecodes)

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

MicroJava	*Java*	
load0	iload0	fload0
load1	iload1	fload1
add	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 expression stack
1 operand	load 7	
2 operands	enter 0, 2	method entry

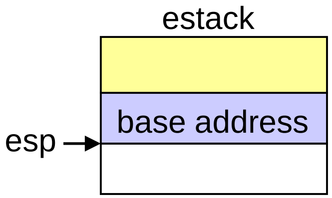
Instruction Set of the μJVM

Addressing modes

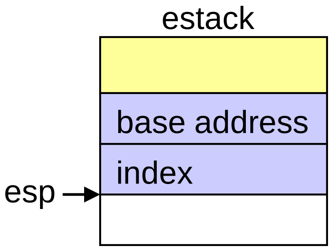
Ways how operands can be accessed

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

Relative



Indexed



Instruction Set of the μJVM

Load/store of local variables

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

operand lengths

b ... byte
s ... short (2 bytes)
w ... word (4 bytes)

Load/store of global variables

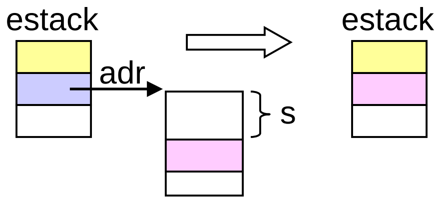
getstatic s	... ..., val	Load static variable push(data[s]);
putstatic s	..., val ...	Store static variable data[s] = pop();

Instruction Set of the μJVM

Load/store of object fields

getfield	s	..., adr ..., val	Load object field adr = pop(); push(heap[adr+s]);

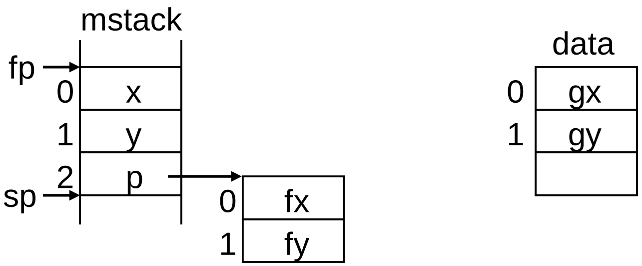
putfield	s	..., adr, val
		...
			Store object field
			val = pop(); adr = pop();
			heap[adr+s] = val;



Loading constants

const	w	... ..., val	Load constant push(w);
const<n>		... ..., val	Load constant (n = 0..5) push(n);
const_m1		... ..., val	Load minus one push(-1);

Examples: Loading and Storing



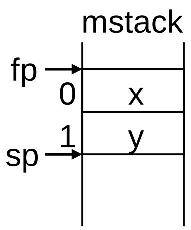
	code	*bytes*	*stack*
x = y;	load1	1	y
	store0	1	-
gx = gy;	getstatic 1	3	gy
	putstatic 0	3	-
p.fx = p.fy;	load2	1	p
	load2	1	p p
	getfield 1	3	p p.fy
	putfield 0	3	-

Instruction Set of the μJVM

Arithmetic

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

Examples: Arithmetic Operations



	code	*bytes*	*stack*
x + y * 3	load0	1	x
	load1	1	x y
	const3	1	x y 3
	mul	1	x y*3
	add	1	x+y*3

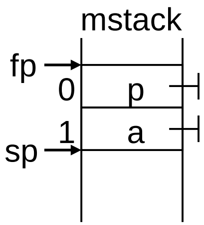
Instruction Set of the μJVM

Object creation

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

newarray b	..., n
	..., adr
		New array
		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

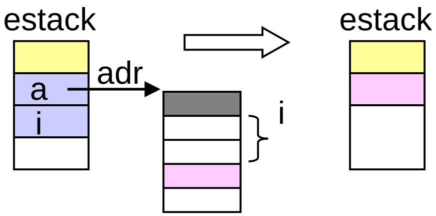


Assume: size(Person) = 4 words

	code	*bytes*	*stack*
Person p = new Person;	new 4	3	p
	store0	1	-
int[] a = new int[5];	const5	1	5
	newarray 1	2	a
	store1	1	-

Instruction Set of the μJVM

Array access



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

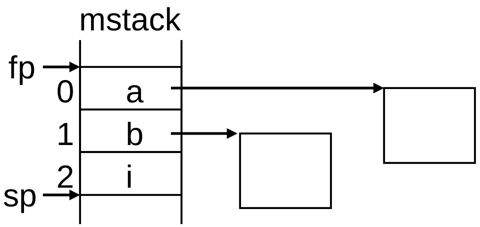
astore	...,adr, i, val
	...
		Store array element
		val = pop(); i = pop(); adr = pop();
		heap[adr+1+i] = val;

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

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 to val;
		heap[adr+1+i/4] = x;

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

Example: Array Access



	code	*bytes*	*stack*
a[i] = b[i+1];	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 μJVM

Stack manipulation

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

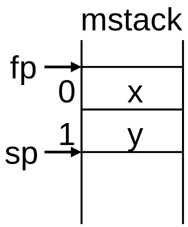
Jumps

jmp s	... ...	Jump unconditionally pc = s;

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

jeq
jne
jlt
jle
jgt
jge

Example: Jumps



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

Instruction Set of the μJVM

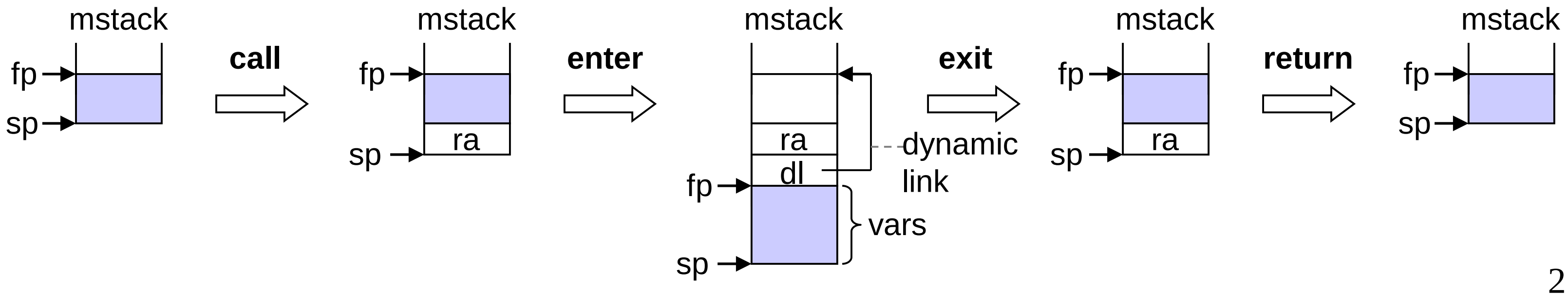
Method call

call	s	... ...	Call method PUSH(pc+3); pc = s;

enter	b1, b2	...
		...
			Enter method
			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();

exit		... ...	Exit method sp = fp; fp = POP();
return		... ...	Return pc = POP();

PUSH and POP work on *mstack*



Instruction Set of the μJVM

Input/output

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

input from System.in
output to System.out

Miscellaneous

trap b	... ...	Throw exception print error message b; stop execution;

Example

void main() int a, b, max, sum; {	0: enter 0, 4
if (a > b)	3: load0 4: load1 5: jle 13
max = a;	8: load0 9: store2 10: jmp 15
else max = b;	13: load1 14: store2
while (a > 0) {	15: load0 16: const0 17: jle 33
sum = sum + a * b;	20: load3 21: load0 22: load1 23: mul 24: add 25: store3
a = a - 1;	26: load0 27: const1 28: sub 29: store0
}	30: jmp 15
}	33: exit 34: return

adresses

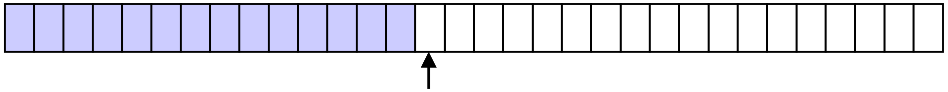
a ...	0
b ...	1
max ...	2
sum ...	3

6. Code Generation

Code Buffer

Data structure

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



buf

pc (= next free position in buf)

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*

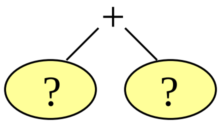
```
Code.put(Code.load0 + 2);
```

6. Code Generation

Operands During Code Generation

Example

we want to add two values



desired code pattern

load operand 1
load operand 2
add

how should that be done?

Depending on the operand kind we must generate different load instructions

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

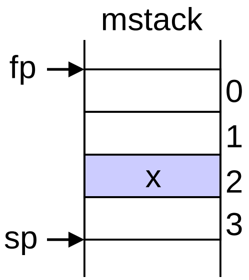
We need a descriptor, which gives us all the necessary information about operands

Operand Decriptors

Descriptors holding information about variables, constants and expressions

Example

Local variable *x* in a stack frame

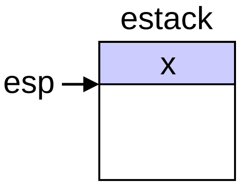


described by the following operand descriptor

kind	Local
adr	2
...	...

=> load2

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



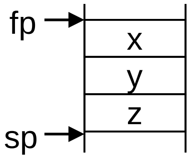
described by the following operand descriptor

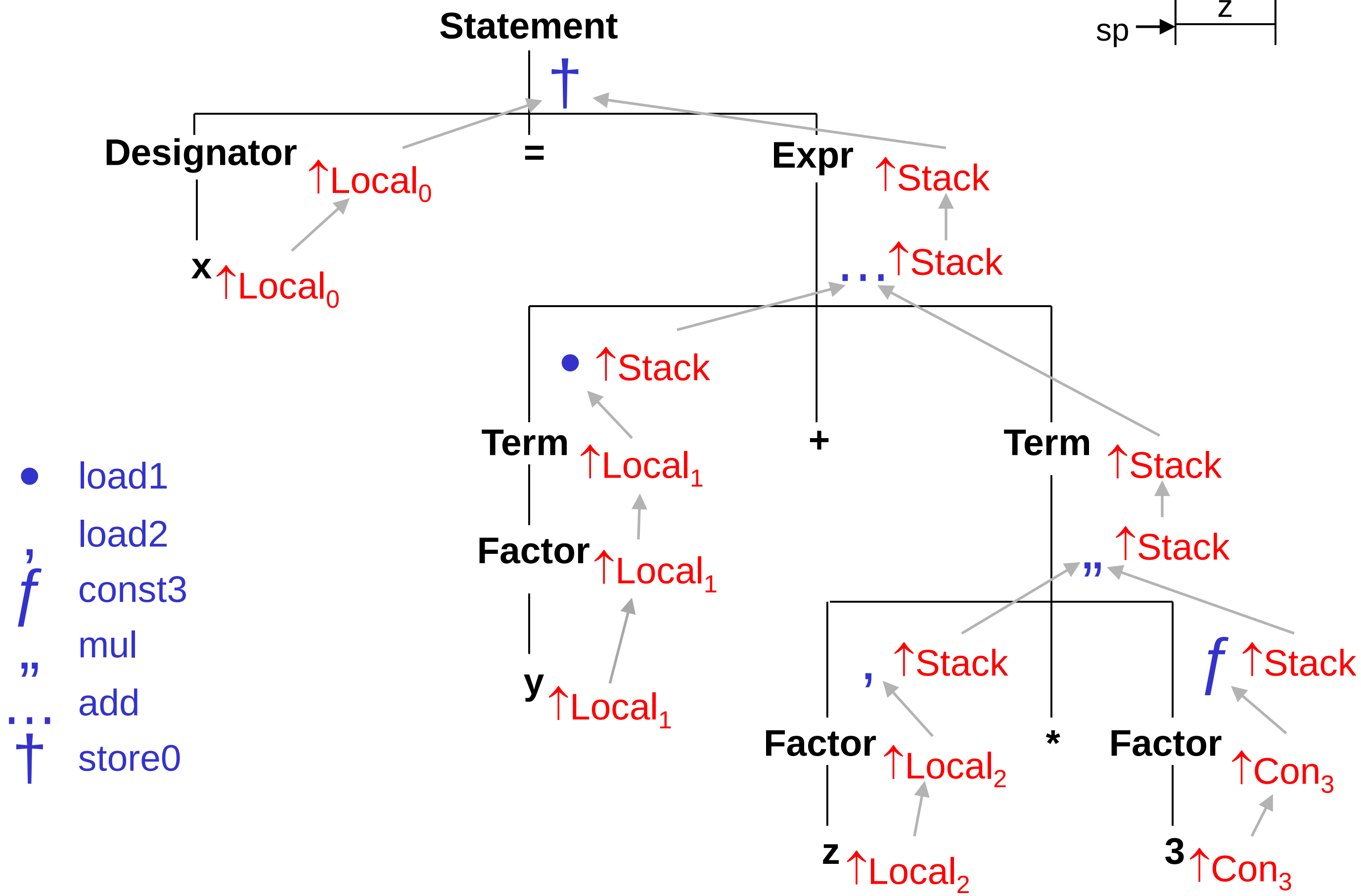
kind	Stack
adr	---
...	...

Example: Processing of Operands

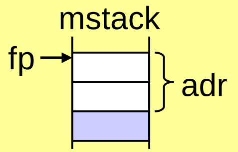
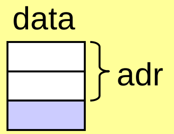
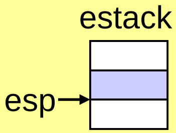
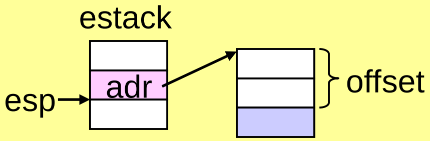
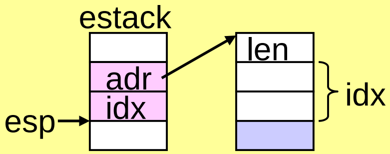
Most parsing methods return operands (as a result of their translation process)

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





Operand Kinds

operand kind	*operand code*	*info about operands*	
constant	**Con** = 0	constant value	
local variable	**Local** = 1	address	
global variable	**Static** = 2	address	
value on the stack	**Stack** = 3	---	
object field	**Fld** = 4	offset	
array element	**Elem** = 5	---	
method	**Meth** = 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

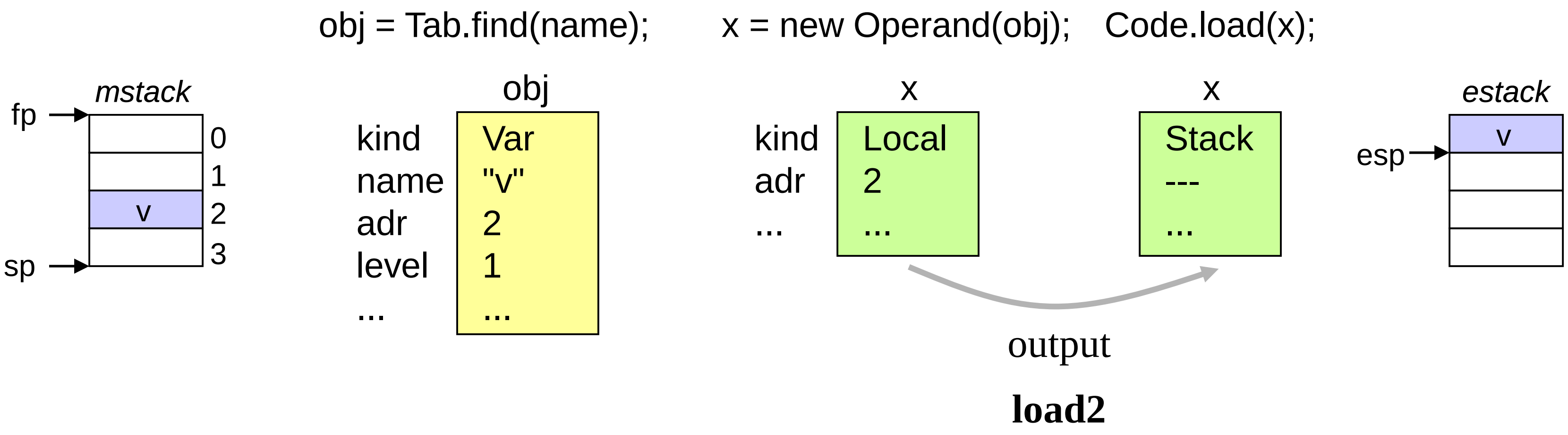
resulting operand is always a *Stack* operand

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



obj = Tab.find(name);   x = new Operand(obj);   Code.load(x);

mstack

fp → 0

1

2: v

sp → 3

obj

kind	Var
name	"v"
adr	2
level	1
...	...

x

kind	Local
adr	2
...	...

x

Stack

...

estack

esp → v

output

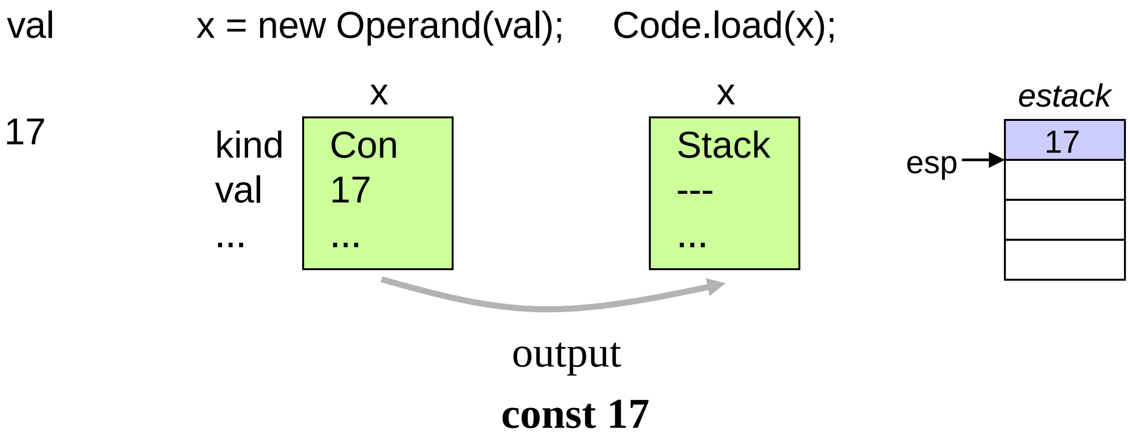
load2

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



val
17

x = new Operand(val);

x
kind | Con
val | 17
... | ...

Code.load(x);

x
Stack

...

estack
esp → 17

output
const 17

Loading Object Fields

var.f

Context conditions (make sure that your compiler checks them)

Designator$_0$ = Designator$_1$ "." ident .

- The type of *Designator*$_1$ must be a class.
- *ident* must be a field of *Designator*$_1$.

Description by an ATG

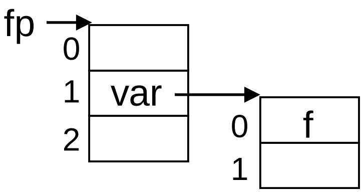
```
Designator <↑x>          (. String name, fName; .)
=  ident <↑name>         (. Obj obj = Tab.find(name);
                            Operand x = new Operand(obj); .)
   { "." ident <↑fName>  (. if (x.type.kind == Struct.Class) {
                               Code.load(x);
                               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*

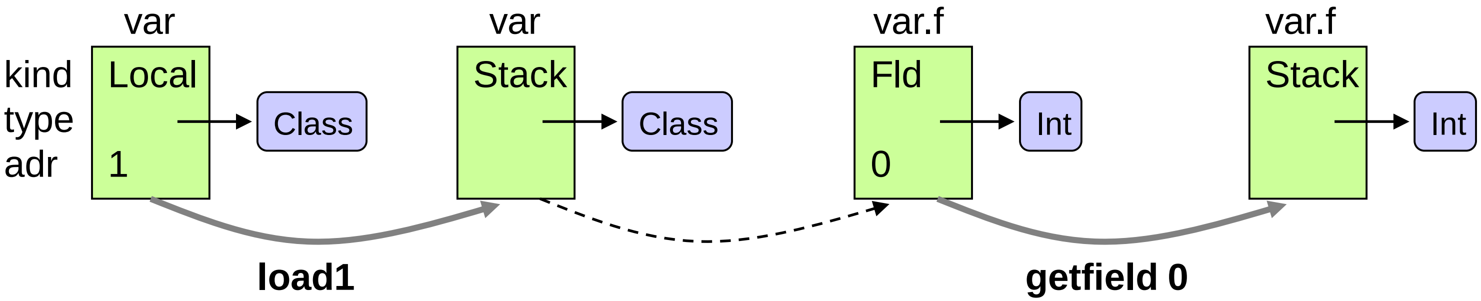
creates a *Fld* operand

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$_0$ = Designator$_1$ "[" Expr "]" .

- The type of *Designator*$_1$ must be an array.
- The type of *Expr* must be *int*.

Description by an ATG

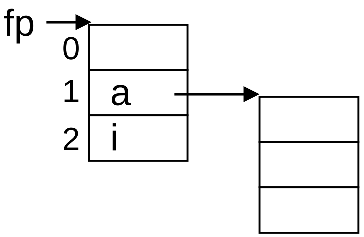
```
Designator <↑x>          (. String name; Operand x, y; .)
=  ident <↑name>         (. Obj obj = Tab.find(name); x = new Operand(obj); .)
   {  ...
   |  "["                (. Code.load(x); .)
      Expr <↑y>          (. if (x.type.kind == Struct.Arr) {
                               if (y.type != Tab.intType) 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"); .)
      "]"
   }.
```

x.kind = Operand.Elem; ← creates an *Elem* operand

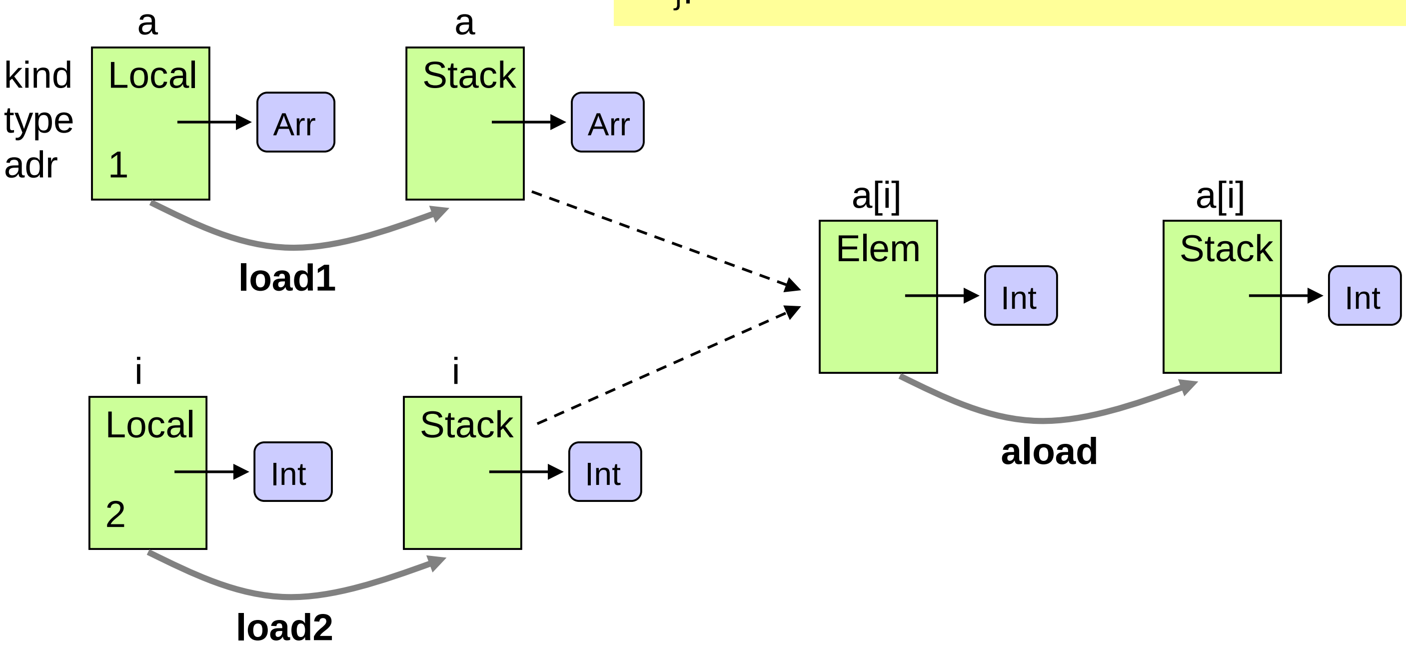
index check is done in the VM

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

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*.

$Expr_0$ = $Expr_1$ Addop Term.

- $Expr_1$ 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; .)
     )                 (. Code.load(x); .)
     Term <↑y>         (. Code.load(y);
                          if (x.type != Tab.intType || y.type != Tab.intType)
                             error("operands must be of type int");
                          Code.put(op); .)
  }.
```

Compiling Terms

$Term_0$ = $Term_1$ Mulop Factor.

- *$Term_1$* 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.

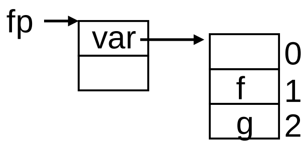
Factor = "new" ident "[" Expr "]".

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

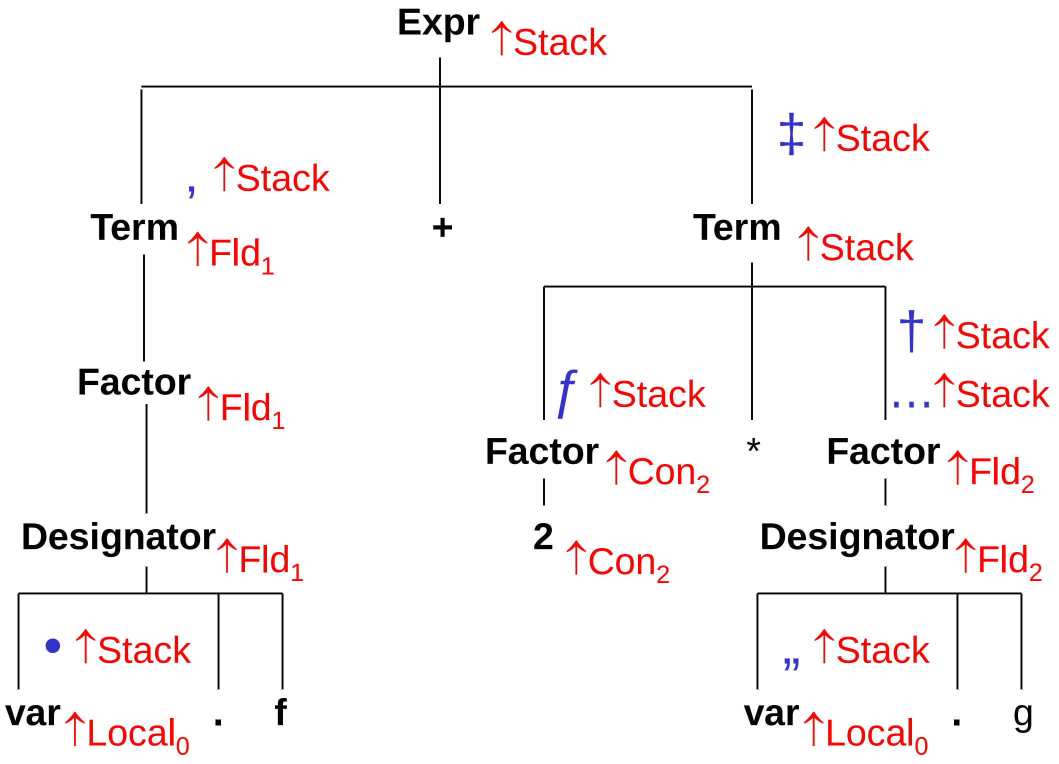
```
Factor <↑x>             (. Operand x; int val; String name; .)
=  Designator <↑x>      // function calls see later
|  number <↑val>        (. x = new Operand(val); .)
|  charCon <↑val>       (. x = new Operand(val); x.type = Tab.charType; .)
|  "(" Expr <↑x> ")"

|  "new" ident <↑name>(. Obj obj = Tab.find(name); Struct type = obj.type; .)

   ( "["                (. if (obj.kind != Obj.Type) error("type expected"); .)
     Expr <↑x> "]"      (. 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(Operand.Stack, 0, type); .)
.
```

Example

var.f + 2 * var.g



•	load0
‚	getfield 1
ƒ	const2
„	load0
…	getfield 2
†	mul
‡	add



6. Code Generation

Code Patterns for Assignments

designator = expr ;

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

localVar = expr;	globalVar = expr;	obj.f = expr;	a[i] = expr;
... *load expr* ... **store adr**$_{\text{localVar}}$	... *load expr* ... **putstatic adr**$_{\text{globalVar}}$	**load obj** ... *load expr* ... **putfield adr**$_{\text{f}}$	**load a** **load i** ... *load expr* ... **astore**

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.load(y);
                             Code.assignTo(x);  // x: Local | Static | Fld | Elem
                         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

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*

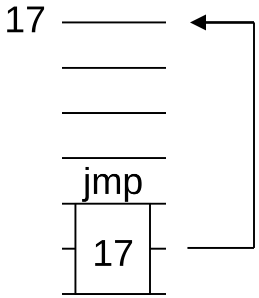
Creation of jump instructions

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

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

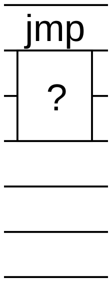
Forward and Backward Jumps

Backward jumps

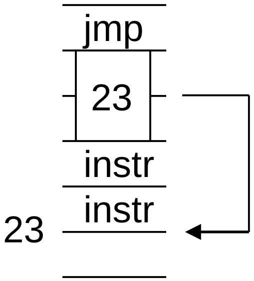


target address is already known
(because the instruction at this position has already been generated)

Forward jumps



target address still unknown
⇒ leave it empty
⇒ remember "fixup address"



patch it when the target address becomes known (fixup)

Conditions

Conditions

if $\underbrace{\text{(a > b)}}_{\textit{Condition}}$ ...

code pattern

$\left.\begin{array}{l}\text{load a}\\ \text{load b}\end{array}\right\}\ \textit{Condition}\uparrow >$

jle ...

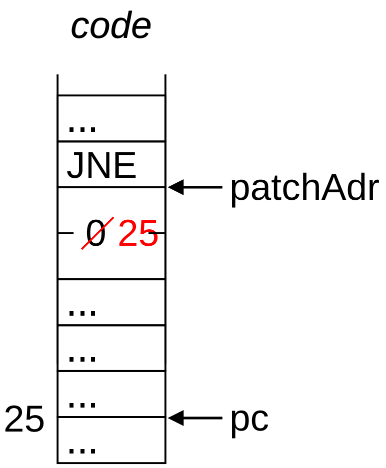
- Problem: the μJVM has no compare instructions
  ⇒ *Condition* cannot generate a compare operation
- therefore, *Condition* has to return the compare operator;
  the comparison is then done in the jump instruction

```
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 uncoditional 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

while Statement

Desired code pattern

```
while
(Condition)

Statement
```

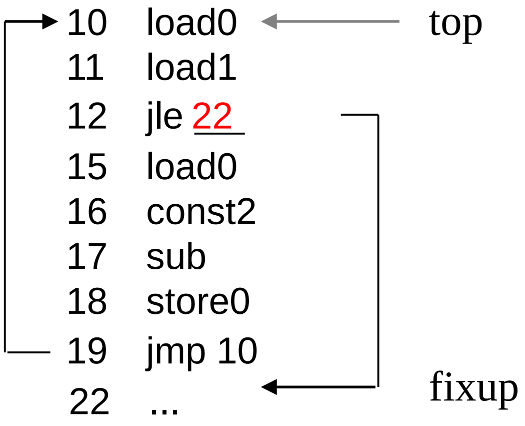
```
top:
        ... code for Condition ...
        falseJump end
        ... code for Statement ...
        jump top
end:    ...
```

Description by an ATG

```
WhileStatement              (. 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

while (a > b) a = a - 2;



```
10   load0      top
11   load1
12   jle 22
15   load0
16   const2
17   sub
18   store0
19   jmp 10
22   ...        fixup
```

if Statement

Desired code pattern

if (Condition) Statement	... *Condition* ... **falseJump end** ... *Statement* ... end: ...

if (Condition) Statement else Statement	... *Condition* ... **falseJump else** ... *Statement* ... **jump end** else: ... *Statement* ... end: ...

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        fixup(adr)
21  store2
22  ...          fixup(adr2)
```

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 *estack*

Description by an ATG

```
Statement                 (. Operand x, y; ... .)
= Designator <↑x>
  (  ActPars <↓x>         (. Code.put(Code.call);
                             Code.put2(x.adr);
                             if (x.type != Tab.noType) Code.put(Code.pop); .)
  |  "=" Expr <↑y> ";"    (. ... .)
  )
| ... .
```

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

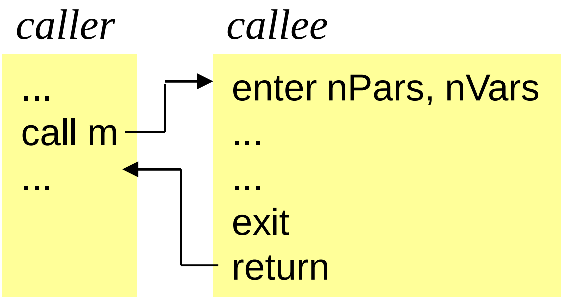
`ord('a')`

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

Description by an ATG

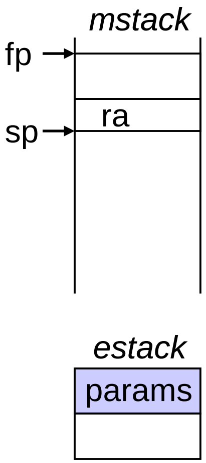
```
Factor <↑x>                (.  Operand x, m; .)
=  Designator <↑x>
   [   ActPars <↓x>        (.  if (x.type == Tab.noType) error("procedure called as a function");
                               if (x.obj == Tab.ordObj || x.obj == Tab.chrObj) ; // nothing
                               else if (x.obj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

Stack Frames



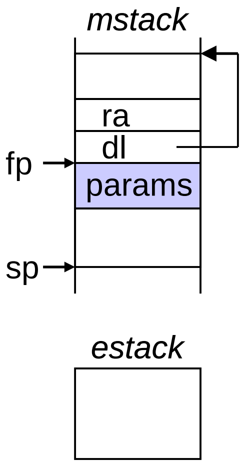
enter ... creates a stack frame
exit ... removes a stack frame

Method entry

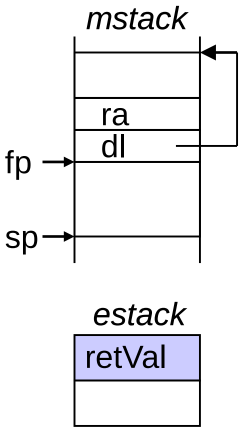


enter nPars, nVars

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

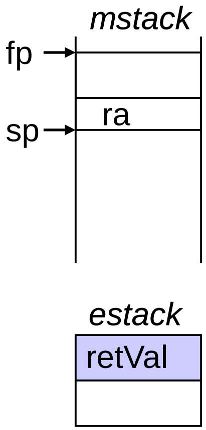


Method exit



exit

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



Method Declaration

```
MethodDecl                   (.  Struct type; String name; int n; .)
= (   Type <↑type>           (. if (type.isRefType()) error("methods may only return int or char"); .)
  |   "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); .)
  Block                      (.  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"); .)
  ")" .
```

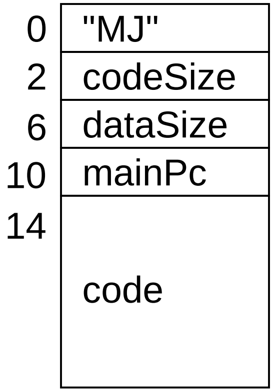
return Statement

```
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.

What you should do in the lab

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 designators (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.

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 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
```