6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

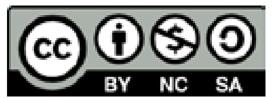
6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

© 2023. This work is licensed under a CC BY-NC-SA 4.0 license
Hanspeter Mössenböck, Institut für Systemsoftware, JKU



Aufgaben der Codeerzeugung

Erzeugung von Maschinenbefehlen

- Auswahl passender Instruktionen
- Auswahl passender Adressierungsarten

Umsetzung von Verzweigungen und Schleifen in Sprünge

Verwalten von Aktivierungssätzen für lokale Variablen

Eventuell Optimierungen

Ausgabe der Objektdatei

Wie geht man vor?

1. **Studieren der Zielmaschine**
   Register, Datenformate, Adressierungsarten, Instruktionen, Befehlsformate, ...
2. **Festlegen von Laufzeitdatenstrukturen**
   Layout von Aktivierungssätzen, globalen Daten, Heapobjekten, Stringkonstantenspeicher, ...
3. **Verwaltung des Codespeichers**
   Bit-Codierung der Instruktionen, Patchen des Codes, ...
4. **Registerverwaltung**
   entfällt in MicroJava, da Stackmaschine
5. **Implementierung der Codeerzeugungsmethoden** (in folgender Reihenfolge)
   - Laden von Variablen, Konstanten und Adressen (in Register oder auf den Stack)
   - Verarbeitung zusammengesetzter Bezeichner (x.y, a[i], ...)
   - Übersetzung von Ausdrücken
   - Verwaltung von Sprüngen und Marken
   - Übersetzung von Anweisungen
   - Übersetzung von Methoden und Parametern

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Architektur der MicroJava-VM (μJVM)

Was ist eine Virtuelle Maschine (VM)?

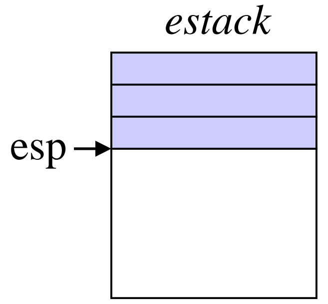
- Eine in Software implementierte CPU
- Befehle werden interpretiert (oder JIT-übersetzt)
- Beispiele: Java-VM, Smalltalk-VM, Pascal P-Code

MicroJava-Programme
μJVM
z.B. Intel-Prozessor

Vorteil: Portabilität von Programmen

Die μJVM ist eine Stackmaschine

- keine Register
- stattdessen *Expression Stack* (auf den Werte geladen werden)



Wortarray (1 Wort = 4 Bytes)
muss nicht groß sein (z.B. 32 Worte entspricht 32 Registern)

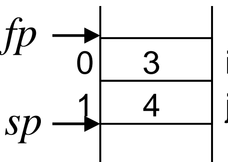
esp ... expression stack pointer

Arbeitsweise einer Stackmaschine

Beispiel

Anweisung **i = i + j * 5;**

Angenommene Werte von *i* und *j*



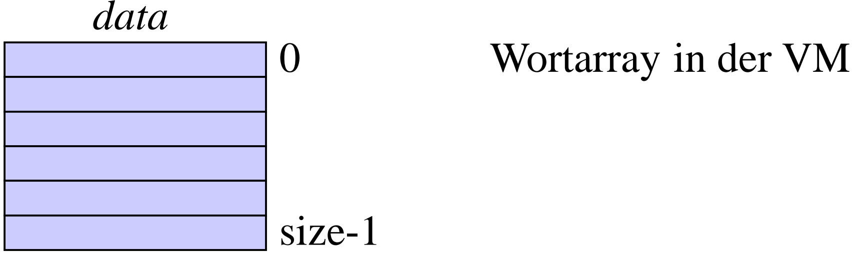
Abarbeitung

Befehle	*Stack*	
load0	3	lade Variable von Adresse 0 (d.h. i)
load1	3 4	lade Variable von Adresse 1 (d.h. j)
const5	3 4 5	lade Konstante 5
mul	3 20	multipliziere die obersten beiden Stackelemente
add	23	addiere die obersten beiden Stackelemente
store0		speichere oberstes Stackelement auf Adresse 0

Am Ende jeder Anweisung ist der Expression Stack wieder leer!

Datenbereiche der μJVM

Globale Variablen

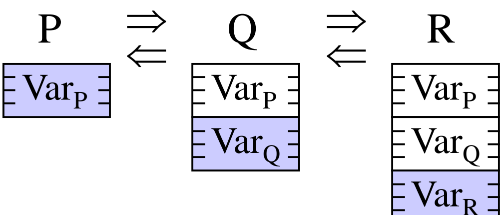


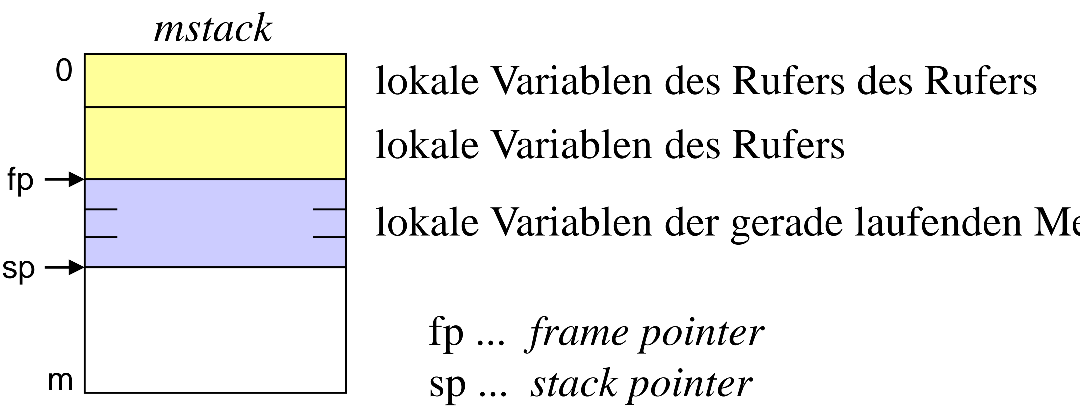
- Bereich fester Größe
- lebt während der gesamten Programmausführung
- jede Variable belegt 1 Wort (4 Bytes)
- Adressierung über Wortnummern
  z.B. *getstatic 2* lädt die Variable mit Adresse 2 von *data* nach *estack*

Datenbereiche der µJVM

Lokale Variablen

- liegen in einem "Aktivierungssatz" (*Stack Frame*)
- jeder Methodenaufruf hat seinen eigenen Stack Frame
- Frames werden kellerartig verwaltet



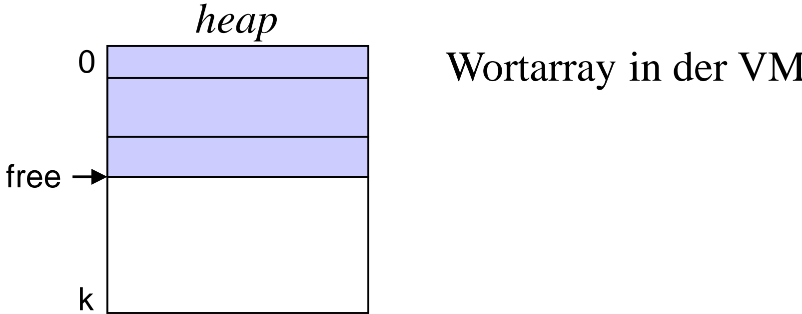


- Variablen werden relativ zu *fp* angesprochen
- Jede Variable belegt 1 Wort (4 Bytes)
- Adressen sind Wortnummern
  z.B. *load0* lädt die Variable mit Offset 0 zu *fp* auf den *estack*

Datenbereiche der μJVM

Heap

- enthält Klassen- und Array-Objekte

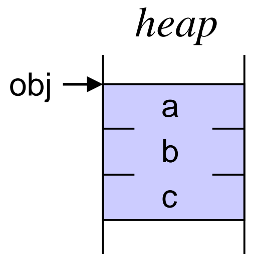


- neue Objekte werden an der Stelle *free* angelegt und *free* wird erhöht; durch die μJVM-Befehle *new* und *newarray*
- Objekte werden in MicroJava nie freigegeben (kein Garbage Collector)
- Zeiger sind Wort-Adressen relativ zum Beginn des Heaps

Datenbereiche der μJVM

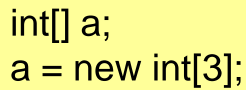
Klassen-Objekte

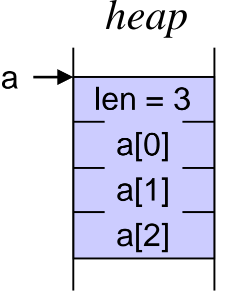
```
class C {
    int a, b;
    char c;
}
C obj = new C;
```



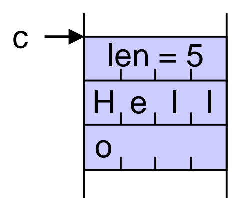
- jedes Feld belegt 1 Wort (4 Bytes)
- Adressierung durch Wortnummern relativ zu *obj*

Array-Objekte





- Länge wird im Array-Objekt gespeichert
- jedes Element belegt 1 Wort (4 Bytes)

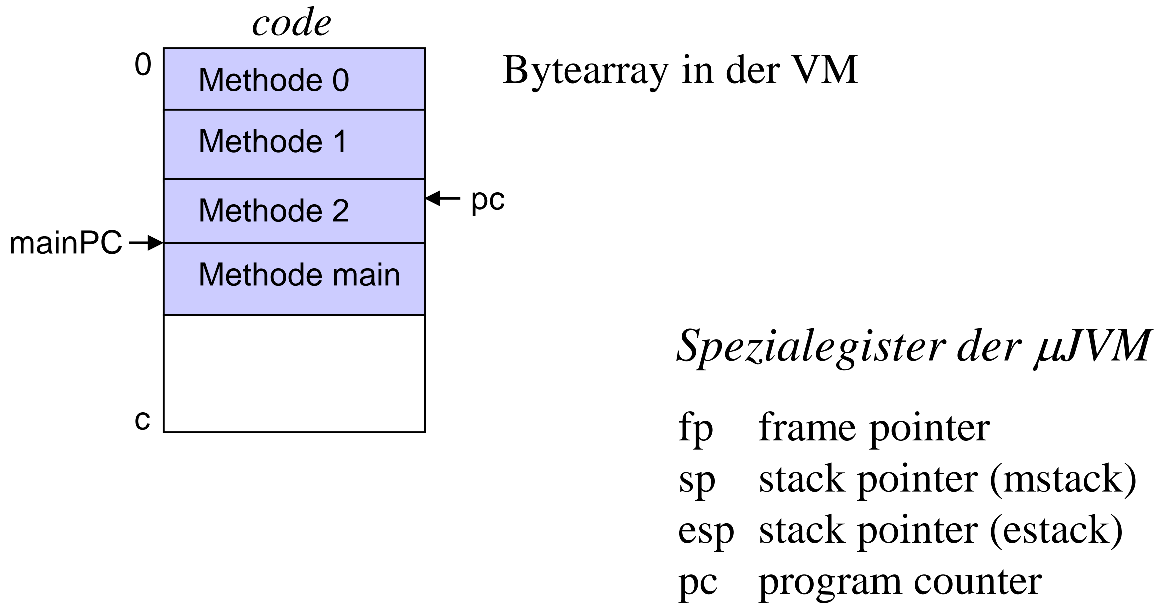


- char-Arrays sind Byte-Arrays
- Größe ist aber ein Vielfaches von 4 Bytes

Codebereich der μJVM

Code

- Byte-Array fixer Größe
- Methoden liegen darin in der Reihenfolge ihrer Deklaration
- *mainPC* zeigt auf *main()*-Methode



Spezialegister der μJVM

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

Instruktionssatz der µJVM

Bytecode (an Java-Bytecode angelehnt)

- sehr kompakt: die meisten Befehle sind nur 1 Byte lang
- ungetypt (in der Java-VM ist der Typ der Operanden in den Befehlen mitcodiert)

MicroJava	*Java*	
load0	iload0	fload0
load1	iload1	fload1
add	iadd	fadd

Grund: Java-Verifier kann so die korrekte Verwendung der Operanden prüfen

Befehlsformat

Sehr einfach im Vergleich zu Intel, ARM oder SPARC

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

opcode ... 1 Byte
operand ... 1, 2 oder 4 Bytes

Beispiele

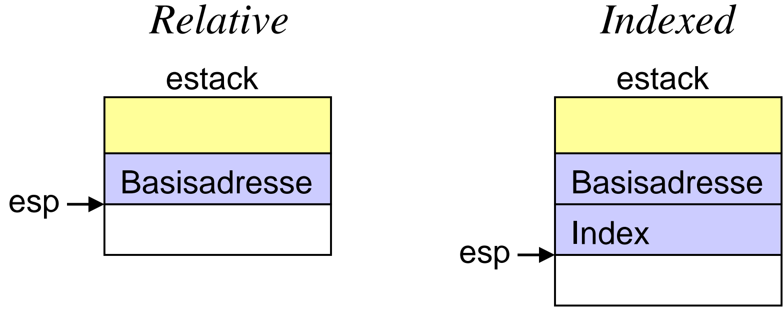
0 Operanden	add	hat zwei implizite Operanden am Stack
1 Operand	load 7	
2 Operanden	enter 0, 2	Methodeneintritt

Instruktionssatz der µJVM

Adressierungsarten

Wie kann man Daten ansprechen? Was bedeuten die Operanden der Befehle?

Adressierungsart	*Beispiel*	
• **Immediate**	const 7	für Konstanten
• **Local**	load 3	für lokale Variablen am *mstack*
• **Static**	getstatic 3	für globale Variablen in *data*
• **Stack**	add	für geladene Werte am *estack*
• **Relative**	getfield 3	für Objektfelder (lade *heap[pop() + 3]*)
• **Indexed**	aload	für Arrayelemente (lade *heap[pop() + pop() + 1]*)



Instruktionssatz der µJVM

Laden und Speichern lokaler Variablen

Instruktion	Stack	Beschreibung
load b	... ..., val	<u>Load</u> push(local[b]);
load*n*	... ..., val	<u>Load</u> (n = 0..3) push(local[n]);
store b	..., val ...	<u>Store</u> local[b] = pop();
store*n*	..., val ...	<u>Store</u> (n = 0..3) local[n] = pop();

Operandenlängen

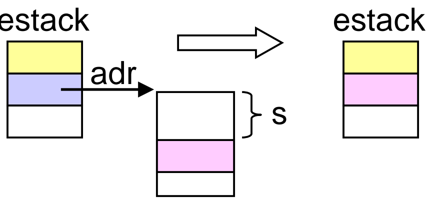
b ... Byte
s ... Short (2 Bytes)
w ... Word (4 Bytes)

Laden und Speichern globaler Variablen

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

Instruktionssatz der μJVM

Laden und Speichern von Objektfeldern



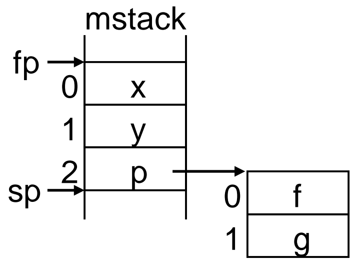
getfield s	..., adr ..., val	Load object field adr = pop(); push(heap[adr+s]);

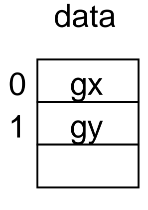
putfield s	..., adr, val
	...
		Store object field
		val = pop(); adr = pop();
		heap[adr+s] = val;

Laden von Konstanten

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);

Beispiele: Laden und Speichern





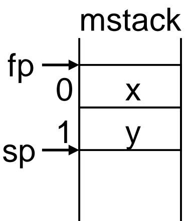
	Code	*Bytes*	*Stack*
x = y;	load1	1	y
	store0	1	-
gx = gy;	getstatic 1	3	gy
	putstatic 0	3	-
p.f = p.g;	load2	1	p
	load2	1	p p
	getfield 1	3	p p.g
	putfield 0	3	-

Instruktionssatz der μJVM

Arithmetik

Instruction	Stack	Description
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 x = pop(); push(pop() / x);
rem	..., val1, val2 ..., val1%val2	Remainder x = pop(); push(pop() % x);
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);
inc b1, b2	... ...	Increment local variable local[b1] = local[b1] + b2;

Beispiele: Arithmetik



	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
x++;	inc 0,1	3	-
x--;	inc 0,-1	3	-

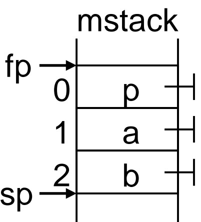
Instruktionssatz der μJVM

Objekterzeugung

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));

Beispiele: Objekterzeugung



Annahme: size(Person) = 4 words

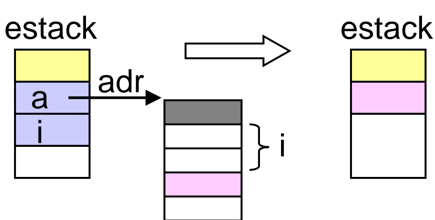
	Code	*Bytes*	*Stack*
Person p = new Person;	new 4	3	p
	store0	1	-
int[] a = new int[5];	const5	1	5
	newarray 1	2	a
	store1	1	-
char[] b = new char[20];	const 20	5	20
	newarray 0	2	b
	store2	1	-

5

20

Instruktionssatz der μJVM

Arrayzugriff



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

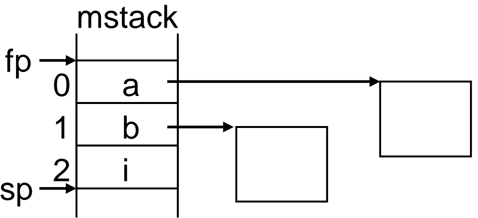
astore	...,adr, i, val
	...
		Store array element
		val = pop(); i = pop(); adr = pop();
		heap[adr+1+i] = val;

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]);

Beispiel: Arrayzugriff



a[i] = b[i+1];

Code	*Bytes*	*Stack*
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	-

Instruktionssatz der μJVM

Stackmanipulation

pop	..., val ...	Remove topmost stack element dummy = pop();
dup	..., val ..., val, val	Duplicate topmost stack element x = pop(); push(x); push(x);

dup2	..., x, y
	..., x, y, x, y
		Duplicate top two stack elements
		y = pop(); x = pop();
		push(x); push(y);
		push(x); push(y);

Sprünge

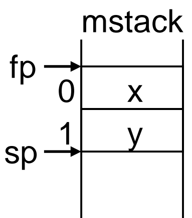
jmp s	... ...	Jump unconditionally pc = pc + s;

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

Sprungdistanz relativ zum Beginn der Instr.

jeq
jne
jlt
jle
jgt
jge

Beispiel: Sprünge



	Code	*Bytes*	*Stack*
if (x > y) ...	load0	1	x
	load1	1	x y
	jle ...	3	-

Instruktionssatz der µJVM

Methodenaufruf

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

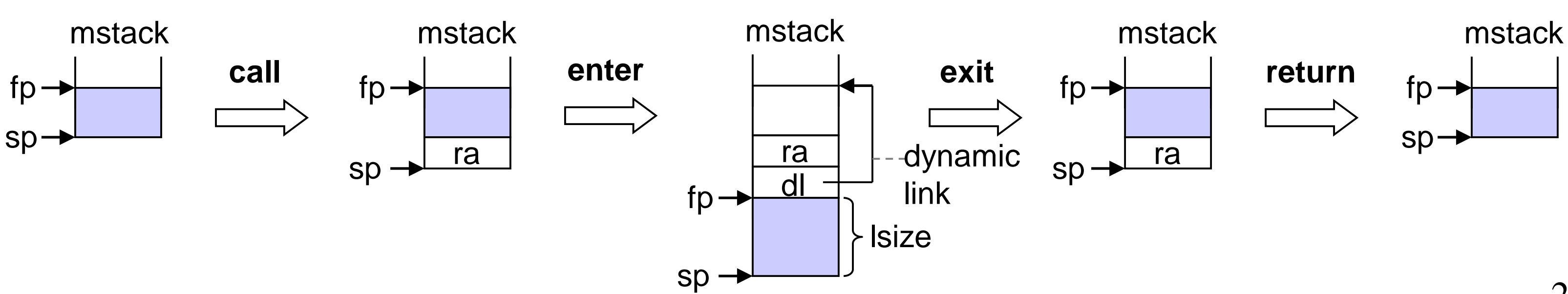
enter b1, b2	...
	...
		Enter method
		psize = b1; lsize = b2; // in words
		PUSH(fp); fp = sp; sp = sp + lsize;
		initialize frame to 0;
		for (i=psize-1; i>=0; i--) local[i] = pop();

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

PUSH und POP arbeiten auf *mstack*

psize ...
Anz. Parameter

lsize ...
Anz. lokaler Var.



Instruktionssatz der µJVM

Ein-/Ausgabe

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

Eingabe von System.in
Ausgabe auf System.out

Gibt *val* in einem Feld mit
w Zeichen rechtsbündig aus

Laufzeitfehler

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

Beispiel

Source	Code
void main() int a, b, max, sum; {	0: enter 0, 4
if (a > b)	3: load0 4: load1 5: jle 8 (=13)
max = a;	8: load0 9: store2 10: jmp 5 (=15)
else max = b;	13: load1 14: store2
while (a > 0) {	15: load0 16: const0 17: jle 15 (=32)
sum = sum + a * b;	20: load3 21: load0 22: load1 23: mul 24: add 25: store3
a--;	26: inc 0, -1
}	29: jmp -14 (=15)
}	32: exit 33: return

Adressen

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

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

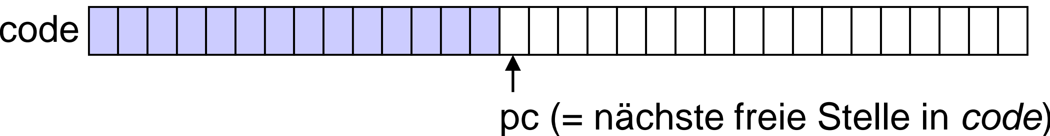
6.8 Ablaufkontrollstrukturen

6.9 Methoden

Codespeicher

Datenstruktur

Byte-Array im Hauptspeicher, da einzelne Instruktionen später noch gepatcht werden müssen.



Instruktionsausgabe

In MicroJava sehr einfach, weil einfaches Befehlsformat

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

    public static void put (int x); {
        code[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);
    }
    ...
}
```

Befehlscodes in Klasse *Code* deklariert

```
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,
    ... ;
```

z.B.: Ausgabe von *load 7*

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

z.B.: Ausgabe von *load2*

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

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Operanden der Codeerzeugung

Beispiel

Es sollen zwei Werte addiert werden

+
? ?

Gewünschtes Codemuster

Lade Operand 1
Lade Operand 2
add

Je nach Operandenart müssen andere Lade-Instruktionen erzeugt werden

Operandenart	*Was muss erzeugt werden?*
• Konstante	const x
• lokale Variable	load a
• globale Variable	getstatic a
• Objektfeld	getfield a
• Arrayelement	aload
• geladener Wert am Stack	---

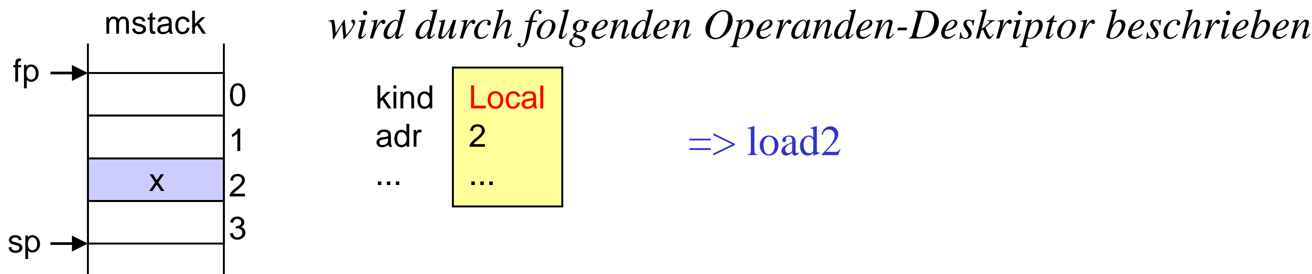
Wir brauchen einen Deskriptor, der uns die Art des Operanden beschreibt

Operanden-Deskriptoren

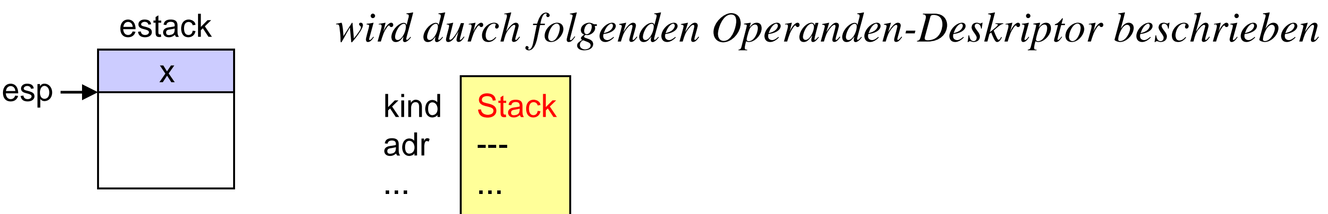
Beschreiben die Art und den Speicherort von Werten

Beispiel

Lokale Variable *x* im Aktivierungssatz



Nach dem Laden mittels *load2* steht der Wert nun am *estack*



Beispiel: Entstehung von Operanden

Die meisten Parsermethoden liefern Operanden (als Ergebnis ihrer Übersetzung)

Beispiel: Übersetzung der Zuweisung `x = y + z * 3;`

fp → x | y | z ← sp

Statement → store0

Designator ↑$Local_0$ = Expr ↑Stack

x ↑$Local_0$

add ↑Stack

Term ↑$Local_1$ → load1 ↑Stack; + ; Term ↑Stack

Factor ↑$Local_1$

y ↑$Local_1$

mul ↑Stack

Factor ↑$Local_2$ → load2 ↑Stack; * ; Factor ↑Con_3 → const3 ↑Stack

z ↑$Local_2$

3 ↑Con_3

- ● load1
- ' load2
- *f* const3
- " mul
- ••• add
- ✝ store0

Arten von Operanden

Operandenart	*Operandencode*	*nötige Infos über Operanden*	
Konstante	**Con** = 0	Konstantenwert	
lokale Variable	**Local** = 1	Adresse	mstack fp adr
globale Variable	**Static** = 2	Adresse	data adr
Wert am Stack	**Stack** = 3	---	estack esp
Objektfeld	**Fld** = 4	Offset	estack esp adr offset
Arrayelement	**Elem** = 5	---	estack esp adr idx len idx
Methode	**Meth** = 6	Adresse, Meth.objekt	

Finden der nötigen Operandenarten

Adressierungsarten
abhängig von Zielmaschine

- Immediate
- Local
- Static
- Stack
- Relative
- Indexed

Objektarten
abhängig von Quellsprache

- Con
- Var
- Type
- Meth

$\cup$

Operandenarten

- Con
- Local
- Static
- Stack
- Fld
- Elem
- Meth

Type-Operanden braucht man in MicroJava nicht, da Typen nicht als Operanden vorkommen können (z.B. keine Type Casts)

Klasse 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;     // Typ des Operanden
    int     val;      // Con: Konstantenwert
    int     adr;      // Local, Static, Fld, Meth: Adresse
    Obj     obj;      // Meth: Methodenobjekt
}
```

Konstruktoren zur Erzeugung von Operanden

```
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;
        case Obj.Type:   error("a type is not a valid operand");
    }
}
```

Erzeugung aus einem Objekt der Symbolliste

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

Erzeugung aus einer Konstanten

Laden von Werten

geg.: Ein Wert, der durch einen Operanden-Deskriptor beschrieben wird (Con, Local, Static, ...)
ges.: Code, um den Wert auf den Stack zu laden

```
public static void load (Operand x) { // Methode der Klasse 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 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)
        case Operand.Meth: error("cannot load a method");
    }
    x.kind = Operand.Stack;
}
```

Fallunterscheidungen

Je nach Operandenart muss ein anderer Ladebefehl erzeugt werden

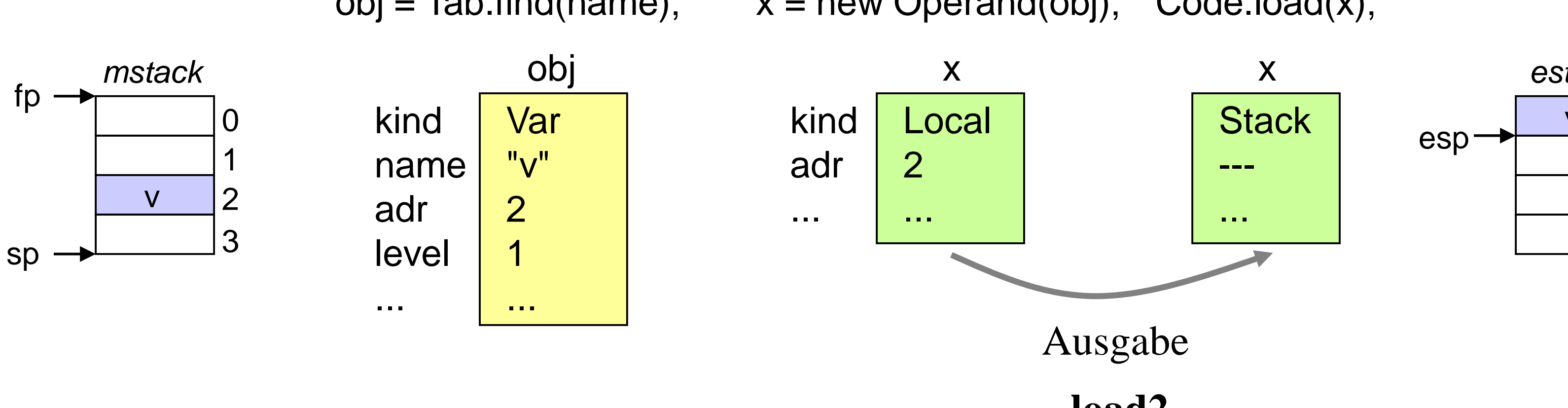
Ergebnis ist immer ein *Stack*-Operand

Beispiel: Laden einer Variablen

Beschreibung durch eine 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
                    .)
| ... .
```

Visualisierung

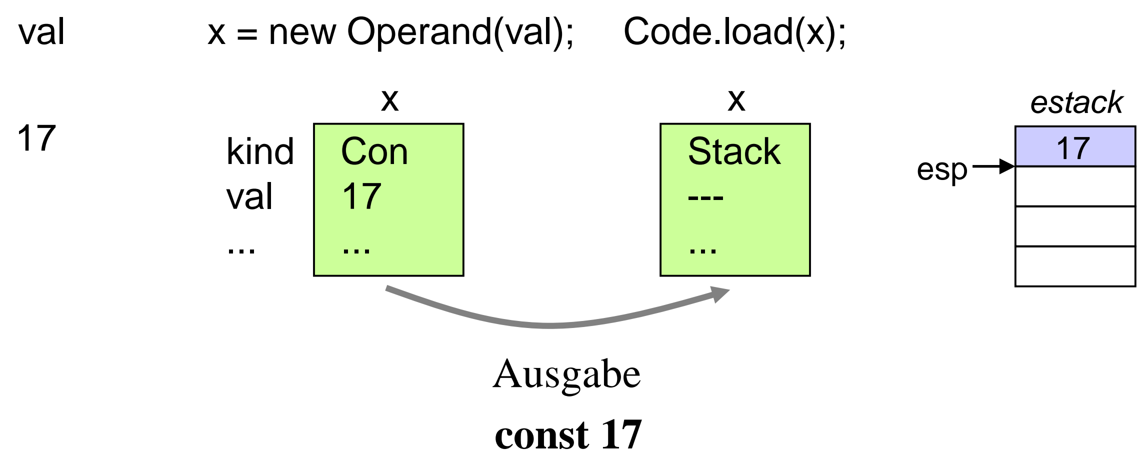


Beispiel: Laden einer Konstanten

Beschreibung durch eine ATG

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

Visualisierung



Laden eines Objektfeldes

var.f

Kontextbedingungen (sicherstellen, dass sie im Compiler geprüft werden)

$Designator_0$ = $Designator_1$ "." ident .

- Der Typ von *Designator*$_1$ muss eine Klasse sein.
- *ident* muss ein Feld von *Designator*$_1$ sein.

Beschreibung durch eine ATG

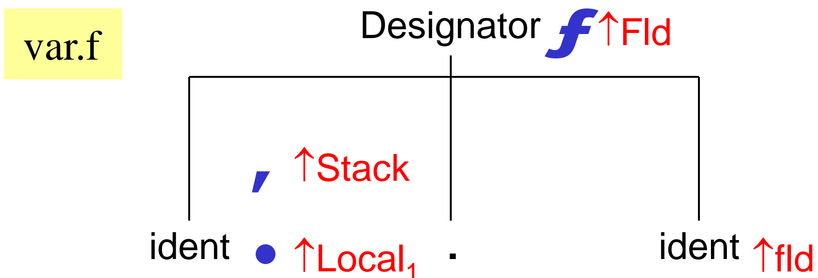
```
Designator <↑x>          (. String name, fname; .)
= ident <↑name>          (. Obj obj = Tab.find(name);
                            Operand x = new Operand(obj); .)
  { "." ident <↑fname>   (. if (x.type.kind == Struct.Class) {
                              Code.load(x);
                       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

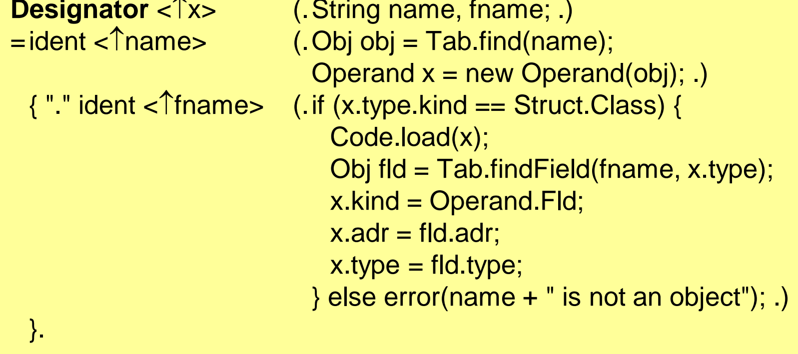
sucht *name1* in der Feldliste von *x.type*

erzeugt einen *Fld*-Operanden

Beispiel: Laden eines Objektfeldes

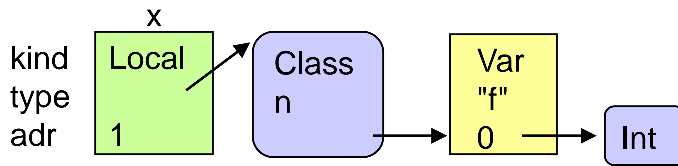
var.f

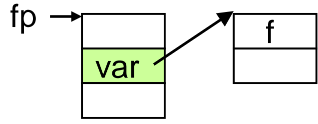




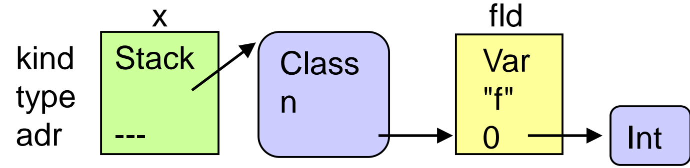
```
Designator <↑x>          (. String name, fname; .)
= ident <↑name>          (. Obj obj = Tab.find(name);
                            Operand x = new Operand(obj); .)
  { "." ident <↑fname>   (. if (x.type.kind == Struct.Class) {
                              Code.load(x);
                       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

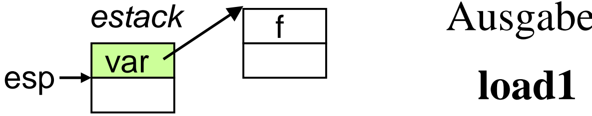
- Operand x = new Operand(varObj);



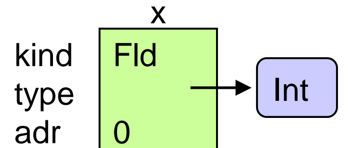


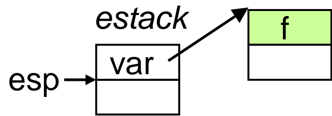
- Code.load(x);



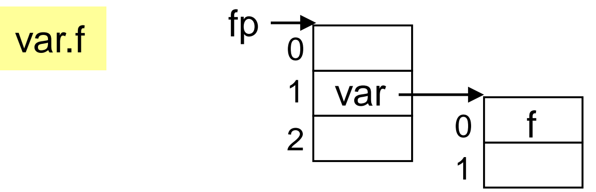


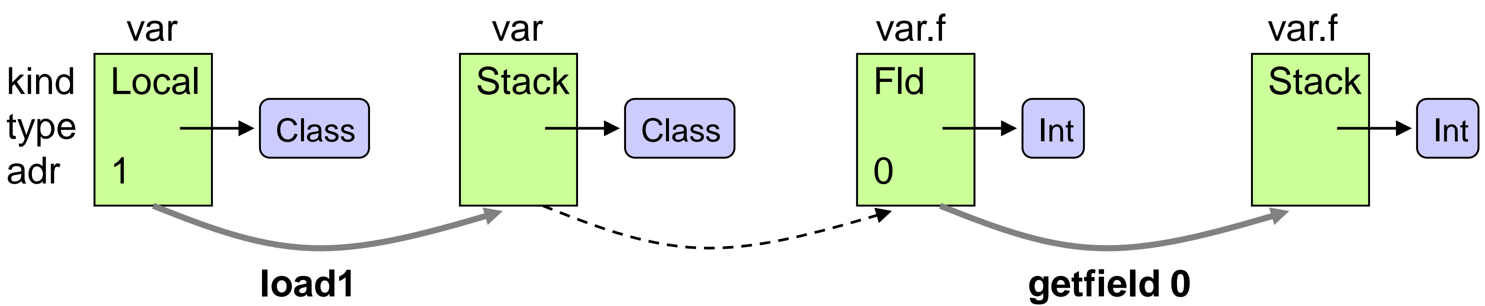
- erzeuge aus *x* und *fld* einen *Fld*-Operanden



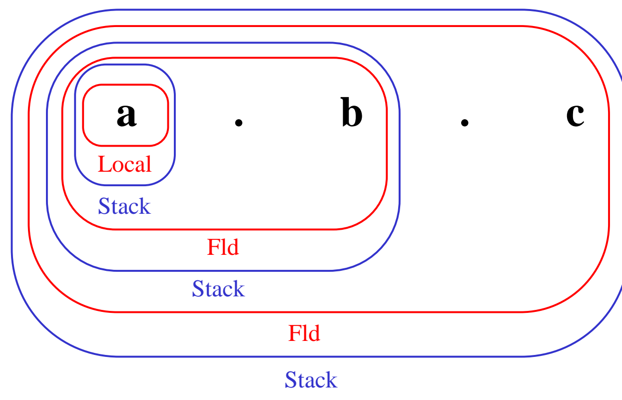


Abfolge der Operanden (1)





Abfolge der Operanden (2)



load a
getfield b
getfield c

Laden eines Arrayelements

a[i]

Kontextbedingungen

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

- Der Typ von *Designator*$_1$ muss ein Array sein.
- Der Typ von *Expr* muss *int* sein.

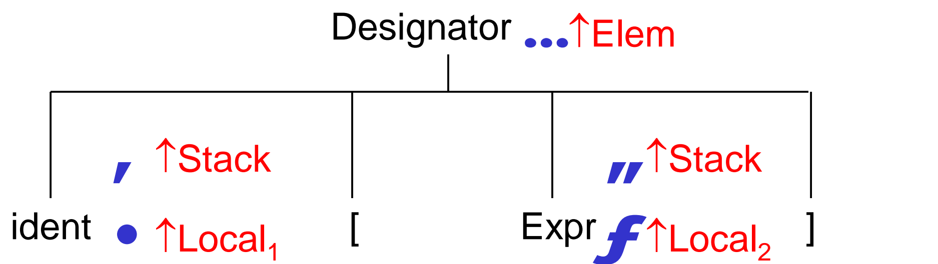
Beschreibung durch eine ATG

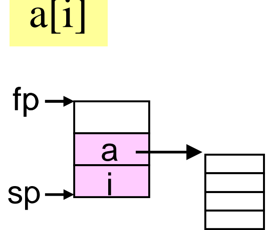
```
Designator <↑x>         (. String name; Operand x, y; .)
= ident <↑name>         (. Obj obj = Tab.find(name); x = new Operand(obj); .)
  { ...
  | "["                 (. Code.load(x); .)
    Expr <↑y>           (. if (x.type.kind == Struct.Arr) {
                                if (y.type != Tab.intType)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

erzeugt einen *Elem*-Operanden (→ x.kind = Operand.Elem;)

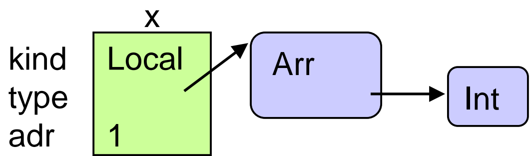
Indexprüfung findet in der VM statt

Beispiel: Laden eines Arrayelements

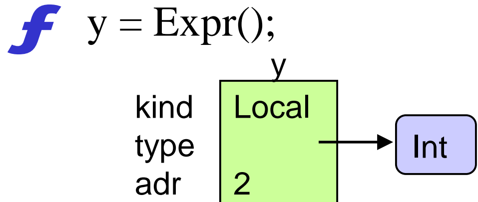




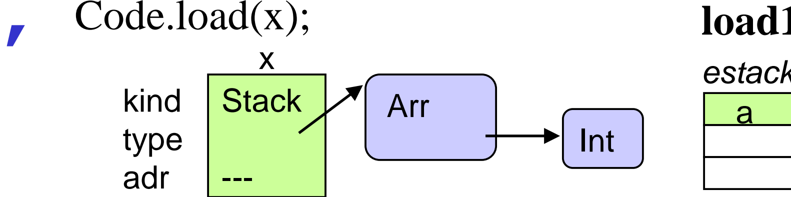
● Operand x = new Operand(obj);



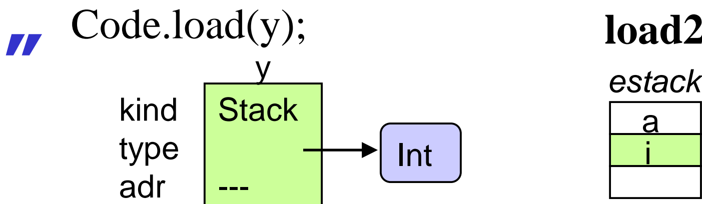
ƒ y = Expr();



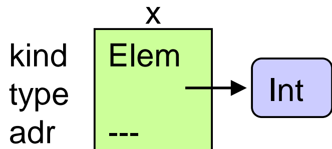
Code.load(x);



Code.load(y);



... erzeuge aus *x* einen *Elem*-Operanden



Abfolge der Operanden (1)

a[i]

fp → 0, 1 a, 2 i

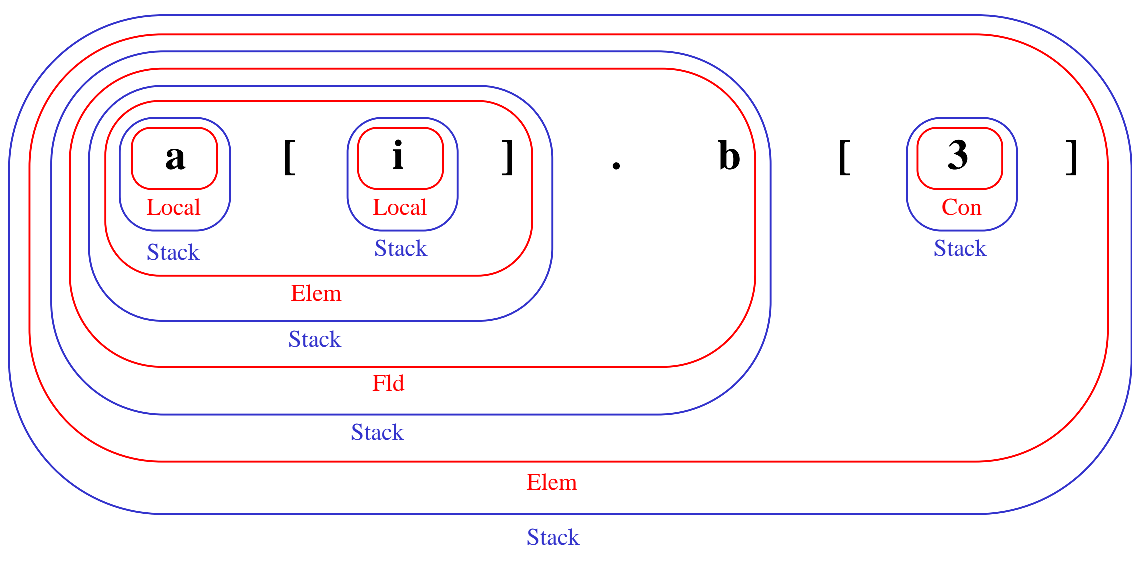
	kind	type	adr
a	Local	Arr	1
a	Stack	Arr	
i	Local	Int	2
i	Stack	Int	
a[i]	Elem	Int	
a[i]	Stack	Int	

load1: a (Local) → a (Stack)

load2: i (Local) → i (Stack)

aload: a[i] (Elem) → a[i] (Stack)

Abfolge der Operanden (2)



load a
load i
aload
getfield b
const3
aload

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Übersetzung von Ausdrücken

Gewünschtes Schema für x + y + z

lade x
lade y
add
lade z
add

Kontextbedingungen

Expr = "-" Term.
• *Term* muss vom Typ *int* sein.

$Expr_0$ = $Expr_1$ AddOp Term.
• $Expr_1$ und *Term* müssen vom Typ *int* sein.

Beschreibung durch eine 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); .)
  }.
```

Übersetzung von Term

$Term_0$ = $Term_1$ MulOp Factor.
• *$Term_1$* und *Factor* müssen vom Typ *int* sein.

```
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); .)
    }.
```

Übersetzung von Factor

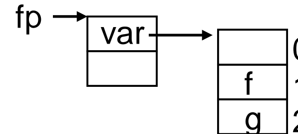
Factor = "new" ident.
• *ident* muss eine Klasse bezeichnen.

Factor = "new" ident "[" Expr "]".
• *ident* muss einen Typ bezeichnen. • Der Typ von *Expr* muss *int* sein.

```
Factor <↑x>                (. Operand x; int val; String name; .)
=  Designator <↑x>         // Funktionsaufrufe siehe später
|  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(); x.kind = Operand.Stack; x.type = type; .)
.
```

Beispiel

var.f + 2 * var.g



- load0
- getfield 1
- const2
- load0
- getfield 2
- mul
- add

Expr ↑Stack

Term ↑Stack ↑Fld_1

\+

Term ↑Stack ↑Stack

Factor ↑Fld_1

Factor ↑Con_2 ↑Stack

*

Factor ↑Fld_2 ↑Stack ↑Stack

Designator ↑Fld_1

2 ↑Con_2

Designator ↑Fld_2

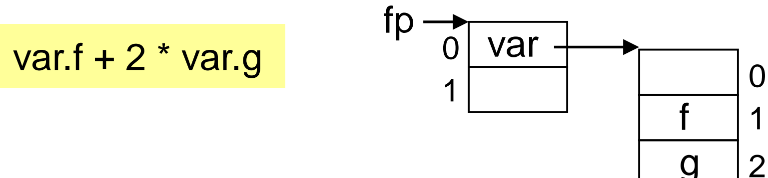
↑Stack

var ↑$Local_0$. f

↑Stack

var ↑$Local_0$. g

Abfolge der Operanden



var: kind Local, type → Class, adr 0 —**load0**→ var: Stack → Class - - → var.f: Fld → Int, 1 —**getfield 1**→ var.f: Stack → Int

2: kind Con, type → Int, val 2 —**const2**→ 2: Stack → Int

var: kind Local, type → Class, adr 0 —**load0**→ var: Stack → Class - - → var.g: Fld → Int, 2 —**getfield 2**→ var.g: Stack → Int

2 (Stack) and var.g (Stack) —**mul**→ 2 * var.g: Stack → Int

var.f (Stack) and 2 * var.g (Stack) —**add**→ var.f + 2 * var.g: Stack → Int

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Codemuster für Zuweisungen

designator = expr ;

4 Fälle, je nach Art der linken Seite

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

Die blauen Instruktionen werden bereits von *Designator* erzeugt!

Übersetzung von Zuweisungen

Kontextbedingung

Statement = Designator "=" Expr ";".

- *Designator* muss eine Variable, ein Arrayelement oder ein Objektfeld bezeichnen.
- Der Typ von *Expr* muss mit dem Typ von *Designator* zuweisungskompatibel sein.

Beschreibung durch eine ATG

```
Assignment            (. Operand x, y; .)
= Designator <↑x>     // dabei wird eventuell schon Code erzeugt
  "=" Expr <↑y>       (. Code.load(y);
                         if (y.type.assignableTo(x.type))
                            Code.assign(x);  // x: Local | Static | Fld | Elem
                         else
                            error("incompatible types in assignment");
                         .)
  ";".
```

Zuweisungskompatibilität

y ist zuweisungskompatibel mit *x*, wenn

- gleiche Typen (*x.type* == *y.type*) oder
- Arrays mit gleichen Elementtypen oder
- *x* hat einen Referenztyp (Klasse oder Array) und *y* ist *null*

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Bedingte und unbedingte Sprünge

Unbedingte Sprünge

jmp offset

Bedingte Sprünge

... *load operand1* ...
... *load operand2* ...
jeq offset

if (operand1 == operand2) jmp offset

jcc	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 Klasse *Code*

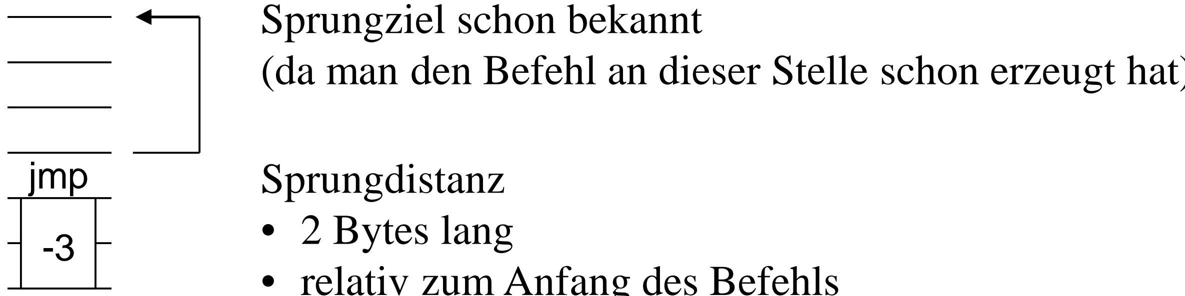
Erzeugung von Sprungbefehlen

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

```
Code.put(Code.jcc + operator);
Code.put2(offset);
```

Vorwärts- und Rückwärtssprünge

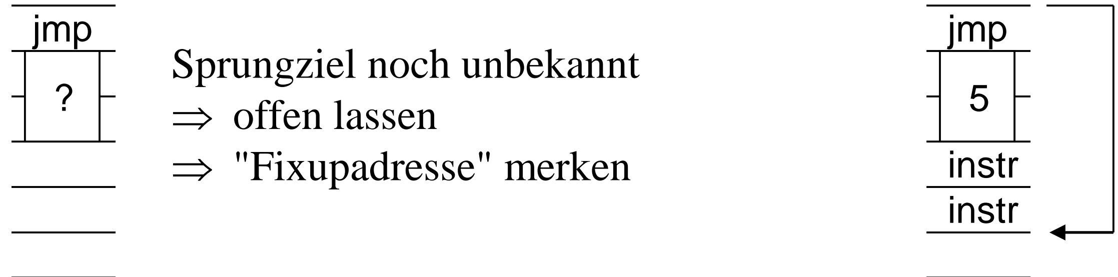
Rückwärtssprünge



Sprungziel schon bekannt
(da man den Befehl an dieser Stelle schon erzeugt hat)

Sprungdistanz
- 2 Bytes lang
- relativ zum Anfang des Befehls

Vorwärtssprünge



Sprungziel noch unbekannt
⇒ offen lassen
⇒ "Fixupadresse" merken

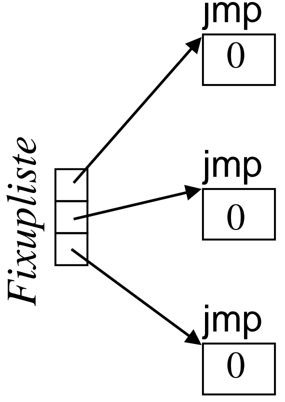
nachtragen, wenn Zieladresse bekannt wird
(Fixup)

Vorwärtssprünge zum gleichen Sprungziel

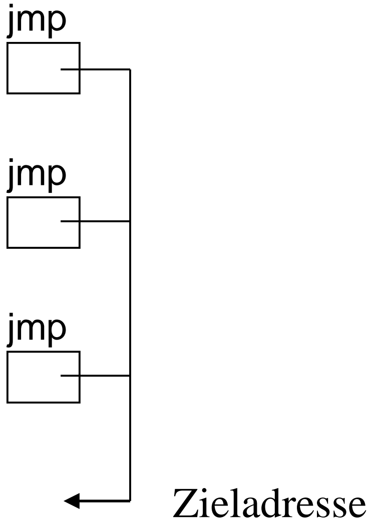
Wo kommt das vor?

- breaks in Schleife oder in switch-Anweisung
- Boolesche Ausdrücke, die mit && und || verknüpft sind (siehe später)

Offene Sprünge zum gleichen Sprungziel werden in einer *Fixupliste* gehalten



Nach Bekanntwerden des Sprungziels wird die Fixupliste aufgelöst



Klasse Label

Verwaltung von Sprungmarken samt Fixuplisten

```
class Label {
    Label() {...}            // erzeugt eine neue noch undefinierte Marke
    void here() {...}        // definiert die Marke an der momentanen pc-Position
    void putAdr() {...}      // gibt Sprungdistanz zu dieser Marke an der pc-Position aus
}
```

Benutzung

```
Label label = new Label();
...
Code.put(Code.jmp);          // Sprung zu noch undefinierter Marke
label.putAdr();              // Fixupadresse wird in label gespeichert
..
label.here();                // Sprung zu label führt hierher
```

Implementierung der Klasse Label

Interner Zustand einer Marke

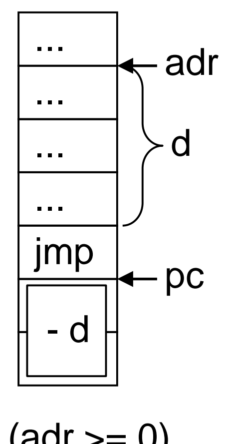
```
class Label {
    private int adr;                             // adr >= 0:  Adresse des Labels; bereits definiert
                                                 // adr < 0:   Label noch undefiniert
    private ArrayList<Integer> fixupList;        // Fixupadressen
    ...
}
```

Konstruktor

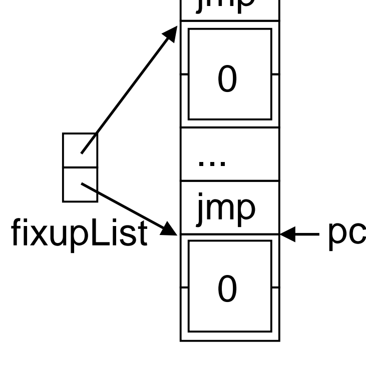
```
public Label() { adr = -1; fixupList = new ArrayList<Integer>(); }
```

putAdr

```
public void putAdr() {
    if (adr >= 0)
        Code.put2(adr - (Code.pc-1));
    else {
        fixupList.add(Code.pc);
        Code.put2(0);
    }
}
```



if (adr >= 0) ...

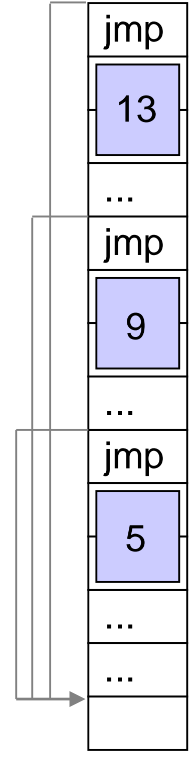


if (adr < 0) ...

Implementierung der Klasse Label

here

```
public void here() {
    if (adr >= 0) error("label defined twice");
    for (int pos: fixupList) {
        Code.put2(pos, Code.pc - (pos-1));
    }
    adr = Code.pc;
}
```



Zusätzliche Methode der Klasse Code

Code.put2(pos, val);   schreibt *val* (2 Bytes) auf Adresse *pos* in den Codespeicher

Bedingungen

Conditions

if (a > b) ...

Condition

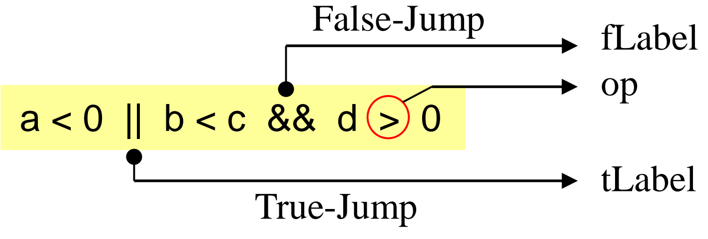
Codemuster

```
load a
load b
jle ...
```

- Problem: Es gibt in der µJVM keinen Vergleichsbefehl
  Vergleich findet erst in Sprungbefehl statt
- *Condition* kann daher den Vergleich noch nicht durchführen
  => liefert stattdessen den Vergleichs<u>operator</u>

Cond-Operand

Condition liefert eine *neue Operanden-Art*
mit folgendem Inhalt:



- Vergleichsoperator: eq, ne, lt, le, gt, ge
- Labels (Fixuplisten) für Sprünge aus der Bedingung heraus
  - tLabel: für True-Jumps
  - fLabel: für False-Jumps

} nötig bei Bedingungen, die mit && und || zusammengesetzt sind

True-Jumps und False-Jumps

True-Jump	springt, wenn die Bedingung wahr ist	a > b	⇒	jgt ...
False-Jump	springt, wenn die Bedingung falsch ist	a > b	⇒	jle ...

Cond-Operanden

```
class Operand {
   static final int Con=0, Local=1, Static=2, Stack=3, Fld=4, Elem=5, Meth=6, Cond=7;

   int     kind;     // Con, Local, Static, ...
   Struct  type;     // Typ des Operanden
   int     val;      // Con: Konstantenwert
   int     adr;      // Local, Static, Fld, Meth: Adresse
   Obj     obj;      // Meth: Methodenobjekt
   int     op;       // Cond: Operator
   Label   tLabel;   // Zielmarke (Fixupliste) für True-Jumps
   Label   fLabel;   // Zielmarke (Fixupliste) für False-Jumps
}
```

Zusätzlicher Konstruktor (u.a. für Cond-Operanden)

```
public Operand (int kind, int val, Struct type) {
   this.kind = kind; this.val = val; this.type = type;
   if (kind == Cond) {
      op = val;
      tLabel = new Label();
      fLabel = new Label();
   }
}
```

Erzeugung unbedingter Sprünge

Mit Hilfsmethode in Klasse *Code*

```
class Code {
    ...
    public static void jump(Label lab) {
        put(jmp); lab.putAdr();
    }
    ...
}
```

Benutzung

```
Label label = new Label();
...
Code.jump(label);
```

Erzeugung bedingter Sprünge

Mit Hilfsmethoden in Klasse *Code*

```
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};
    ...
    public static void tJump (int op, Label label) {
        put(jcc + op);  // jeq, jne, jlt, jle, jgt, jge
        label.putAdr();
    }

    public static void fJump (int op, Label label) {
        put(jcc + inverse[op]);  // jne, jeq, jge, jgt, jle, jlt
        label.putAdr();
    }
    ...
}
```

Benutzung

```
"if" "(" Condition <↑x> ")"        (.  Code.fJump(x.op, x.fLabel); .)
```

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

while-Anweisung

Gewünschtes Codemuster

while (Condition) Statement	top: ... Code für *Condition* ... **fJump end** ... Code für *Statement* ... **jump top** end: ...

Beschreibung durch eine ATG

```
WhileStatement              (. Operand x; .)
=  "while"                  (. Label top = new Label();
                               top.here(); .)
   "(" Condition <↑x> ")"   (. Code.fJump(x.op, x.fLabel); .)
   Statement                (. Code.jump(top);
                               x.fLabel.here(); .)
   .
```

Beispiel

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

```
10   load0        ← top
11   load1
12   jle 10 (=22)
15   load0
16   const2
17   sub
18   store0
19   jmp -9 (=10)
22   ...          ← x.fLabel
```

if-Anweisung

Gewünschtes Codemuster

```
if
(Condition)

Statement
```

```
      ... Condition ...
      fJump end
      ... Statement ...
end:  ...
```

```
if
(Condition)

Statement
else

Statement
```

```
        ... Condition ...
        fJump else
        ... Statement ...
        jump end
else:
        ... Statement ...
end:    ...
```

Beschreibung durch eine ATG

```
IfStatement                 (. Operand x; Label end; .)
= "if"
  "(" Condition <↑x> ")"    (. Code.fJump(x.op, x.fLabel); .)
  Statement
  (  "else"                 (. end = new Label();
                               Code.jump(end);
                               x.fLabel.here(); .)

     Statement              (. end.here(); .)

  |                         (. x.fLabel.here(); .)
  ).
```

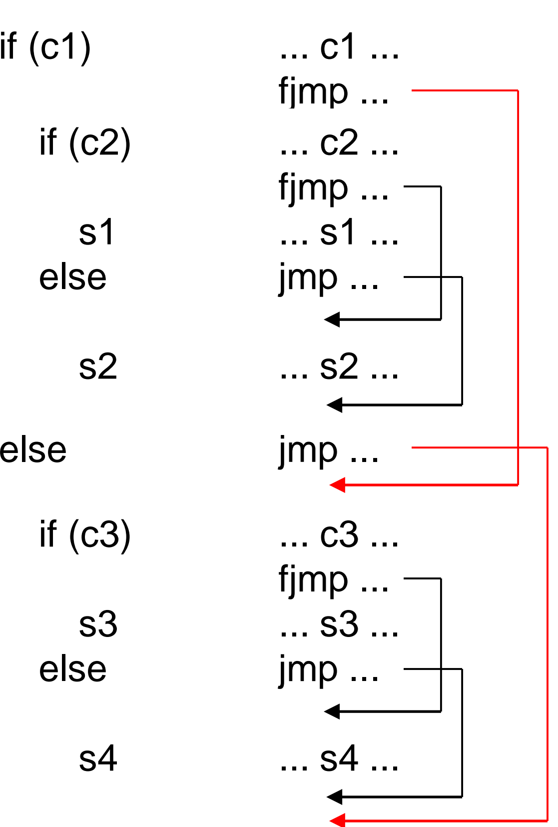
Beispiel

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

```
10   load0
11   load1
12   jle 8 (=20)
15   load0
16   store2
17   jmp 5 (=22)
20   load1        x.fLabel
21   store2
22   ...          end
```

Funktioniert auch bei geschachtelten ifs

```
IfStatement                (. Operand x; Label end; .)
= "if"
  "(" Condition <↑x> ")"   (. Code.fJump(x.op, x.fLabel); .)
  Statement
  (  "else"                (. end = new Label();
                              Code.jump(end);
                              x.fLabel.here(); .)
     Statement             (. end.here(); .)
  |                        (. x.fLabel.here(); .)
  ).
```



break-Anweisung

Aussprung aus Schleifen

- Marke *breakLab* ans Ende jeder Schleife setzen
- bei Auftreten einer break-Anweisung: Code.jump(breakLab);

Geschachelte Schleifen

- Jede Schleife braucht ihr eigenes *breakLab*
- *breakLab* muss mit Hilfe eines Stacks gerettet werden

```
                              (. // global declarations
                                 Label breakLab = null;
                                 Stack<Label> breaks = new Stack<>();
                                 .)
Statement
=  "while"                    (. breaks.push(breakLab);
                                 breakLab = new Label(); ... .)
   "(" Condition <↑x> ")"     (. ... .)
   Statement                  (. ...
                                 breakLab.here();
                                 breakLab = breaks.pop(); .)
|  "break" ";"                (. if (breakLab == null) error("break outside a loop");
                                 Code.jump(breakLab); .)
|  ... .
```

Frage: Was müsste man tun, um *break* mit einer Marke verwenden zu können?

Kurzschlussauswertung Boolescher Ausdrücke

- Boolesche Ausdrücke können mit && und || zusammengesetzt sein
- Berechnung des Ausdrucks wird abgebrochen, sobald sein Ergebnis feststeht

Kleine Änderung in ATG von if-Anweisung

```
IfStatement
=  "if"
   "(" Condition <↑x> ")"      (. Code.fJump(x.op, x.fLabel);
                                  x.tLabel.here(); .)
   Statement                   (. x.fLabel.here(); .)
   .
```

ähnlich in while-Anweisung

Übersetzung Boolescher Ausdrücke

```
CondFactor <↑x>     (. Operand x, y; int op; .)
= 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");
                       x = new Operand(Operand.Cond, op, null);
                    .) .
```

```
CondTerm <↑x>             (. Operand x, y; .)
= CondFactor <↑x>
  { "&&"                  (. Code.fJump(x.op, x.fLabel); .)
    CondFactor <↑y>       (. x.op = y.op; .)
  }.
```

```
Condition <↑x>            (. Operand x, y; .)
= CondTerm <↑x>
  { "||"                  (. Code.tJump(x.op, x.tLabel);
                             x.fLabel.here(); .)
    CondTerm <↑y>         (. x.op = y.op; x.fLabel = y.fLabel; .)
  }.
```

```
a>b && b>c || d<e && e<f

load a
load b
jle ...
load b
load c
jgt ...
load d
load e
jge ...
load e
load f
```

6. Codeerzeugung

6.1 Überblick

6.2 Die MicroJava VM

6.3 Codespeicher

6.4 Operanden

6.5 Ausdrücke

6.6 Zuweisungen

6.7 Sprünge und Marken

6.8 Ablaufkontrollstrukturen

6.9 Methoden

Prozeduraufruf

Codemuster

```
m(a, b);        load a
                load b
                call m
```

Parameter werden am *estack* übergeben

Beschreibung durch eine ATG

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

Funktionsaufruf

Codemuster

```
c = m(a, b);    load a
                load b
                call m
                store c
```

load a — Parameter werden am *estack* übergeben

store c — Funktionswert kommt am *estack* zurück

Beschreibung durch eine ATG

```
Factor <↑x>                 (. Operand x; .)
= Designator <↑x>
  [ ActPars <↓x>            (. if (x.type == Tab.noType) error("procedure called as a function");
                               Code.callMethod(x);
                               x.kind = Operand.Stack; .)
  ]
| ... .
```

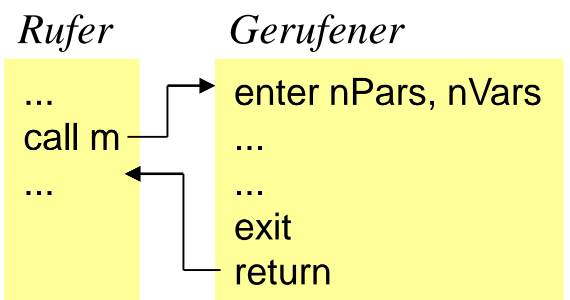
```
static void callMethod (Operand m) {
   if (m.obj == Tab.ordObj || m.obj == Tab.chrObj) ; // nothing
   else if (m.obj == Tab.lenObj)
      Code.put(Code.arraylength);
   else {
      Code.put(Code.call);
      Code.put2(m.adr - (Code.pc-1));
   }
}
```

Standardfunktionen

ord('a')

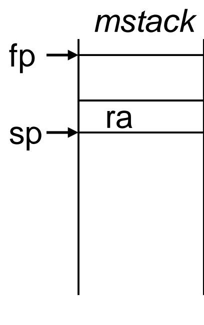
- *ActPars* lädt 'a' auf den *estack*
- geladener Wert bekommt Typ von *ordObj* (= *intType*) und *kind* = *Operand.Stack*

Aktivierungssätze (Frames)



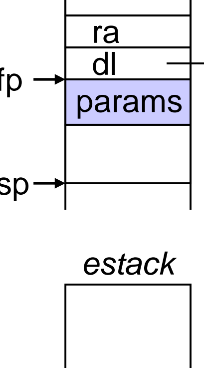
enter ... richtet einen Frame ein
exit ... entfernt ihn wieder

Methodeneintritt

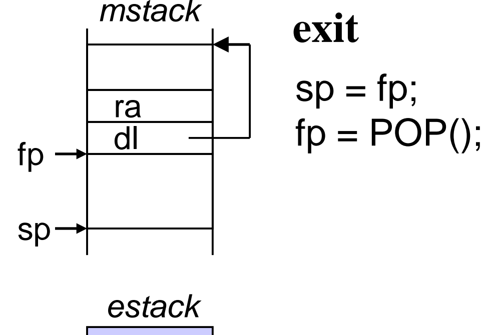


enter nPars, nVars

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

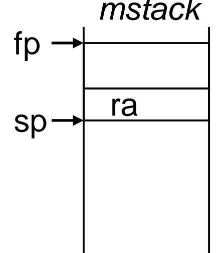


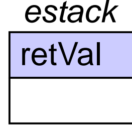
Methodenaustritt



exit

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





Methodendeklaration

```
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(); .)
  .
```

Formale Parameter

- in Symbolliste eintragen (als Variablen des Methoden-Scopes)
- Anzahl zählen

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

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

Aktuelle Parameter

- auf *estack* laden
- prüfen, ob zuweisungskompatibel zu formalen Parametern
- prüfen, ob Parameteranzahl stimmt

```
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");
                             } .)
     { "," Expr <↑ap>     (. Code.load(ap); aPars++;
                             fp = fp.next;
                             if (fp != null) {
                                if (!ap.type.assignableTo(fp.type)) error("parameter type mismatch");
                             } .)
     }
  ]                       (. if (aPars > fPars)
                                error("too many actual parameters");
                             else if (aPars < fPars)
                                error("too few actual parameters"); .)
  ")" .
```

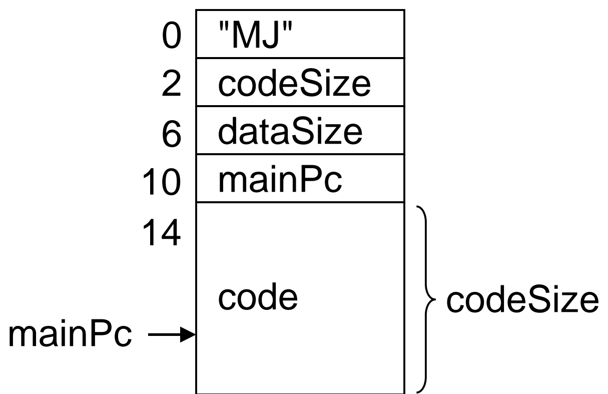
return-Anweisung

```
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_); .)
   ";".
```

Objektdatei

Inhalt in MicroJava

- Ladeinformationen
  - Größe des Codes (in Bytes)
  - Größe des globalen Datenbereichs (in Worten)
  - Adresse der *main*-Methode
- Code



Weitere Informationen in anderen Sprachen (Java, C, Pascal, ...)

- Stringkonstantenspeicher
- Liste exportierter Symbole ({Name Adresse})
- Vorkommen importierter Symbole im Code (Fixup-Information)
  { Name {CodeAdresse} }
- Metainformationen (für Debugger und für dynamisches Nachladen)
- ...