

Lecture 16 — Compiling MiniJava, cont.

- **Compiling while statements**
- **Short-circuit evaluation of boolean expressions**
- **Arrays**
- **Switch statements**
- **V-tables and objects**

Compilation schemes

Methods: $M \rightsquigarrow il$

Statements: $S, m \rightsquigarrow il, m'$

Expressions: $e, loc \rightsquigarrow il$

Compiling while statements

While $(e) S, m \rightsquigarrow$

$[JUMP\ m'] @\ ils @\ ile @ [CJUMP\ loc, m + 1, m''], m''$

$S, m + 1 \rightsquigarrow ils, m'$

$e, loc \rightsquigarrow ile$

(where $m'' = m' + |ile| + 1$)

do S while $(e), m \rightsquigarrow$

$S,$

$e,$

Evaluation of boolean expressions

- MP7 uses strict evaluation of boolean expressions:

if (e) S_1 else S_2 , $m \rightsquigarrow$

$il @ [\text{CJUMP } \text{loc}, m + |il| + 1, m' + 1] @ il_1 @ [\text{JUMP } m''] @ il_2, m''$
 $e, \text{loc} \rightsquigarrow il$
 $S_1, m + |il| + 1 \rightsquigarrow il_1, m'$
 $S_2, m' + 1 \rightsquigarrow il_2, m''$

$x=e, m \rightsquigarrow il @ [\text{MOV}(\text{addr } x, \text{loc})], m + |il| + 1$
(x a variable)

OperationT(e_1, bop, e_2), $\text{loc} \rightsquigarrow il_1 @ il_2 @ [\text{BOP } \text{loc}, \text{loc1}, \text{loc2}]$

$e_1, \text{loc1} \rightsquigarrow il_1$

$e_2, \text{loc2} \rightsquigarrow il_2$

Evaluation of boolean expressions (cont.)

```
public int main (int m, int n) {  
    if (m<n & (m < 10 | 10 < n))  
        n = 0;  
    else  
        n=1;  
    return n;  
}
```

0:	LESS	3,1,2		CJUMP	9,8,11
	LOADIMM	4,10	8:	LOADIMM	3,0
	LESS	5,1,4		MOV	2,3
	LOADIMM	6,10		JUMP	13
	LESS	7,6,2	11:	LOADIMM	3,1
	OR	8,5,7		MOV	2,3
	AND	9,3,8	13:	RETURN	2

Short-circuit evaluation of boolean expressions

- The best way to compile boolean expressions is to avoid computing the value of the expression.

$$e, m, t, f \rightsquigarrow_2 il, m'$$

- Some expressions are compiled very simply:

$$\text{True}, m, t, f \rightsquigarrow_2 [\text{JUMP } t], m + 1$$

$$\text{False}, m, t, f \rightsquigarrow_2 [\text{JUMP } f], m + 1$$

$$!e, m, t, f \rightsquigarrow_2 il, m'$$

$$e, m, f, t \rightsquigarrow_2 il, m'$$

Short-circuit evaluation of boolean expressions (cont.)

$$e_1 \& \& e_2, m, t, f \rightsquigarrow_2$$

$$e_1 || e_2, m, t, f \rightsquigarrow_2$$

$$\text{If } (e) S_1 \text{ else } S_2, m \rightsquigarrow$$

Short-circuit evaluation of boolean expressions (cont.)

```
public int main (int m, int n) {  
    if (m<n & (m < 10 | 10 < n))  
        n = 0;  
    else  
        n=1;  
    return n;  
}
```

0:	LESS	3,1,2		CJUMP	7,8,11
	CJUMP	3,2,11	8:	LOADIMM	3,0
2:	LOADIMM	4,10		MOV	2,3
	LESS	5,1,4		JUMP	13
	CJUMP	5,8,5	11:	LOADIMM	3,1
5:	LOADIMM	6,10		MOV	2,3
	LESS	7,6,2	13:	RETURN	2

Arrays in MJ

- Arrays stored in the heap. Contents are integers — values or pointers to heap objects (including arrays).
- Have instruction (not used in MP7):

```
ARRAYREF tgt,src,indx: (p, c, s, h, t, r)  
                    -> (p+1, c, s[i/tgt], h, t, r)
```

- Array indexing:

$a[e]$, **loc** $\rightsquigarrow$

Multi-dimensional arrays in MJ

- A multi-dimensional array is an array that contains pointers to other arrays.
- Array indexing for multi-dimensional arrays:

$e_1[e_2], \text{loc} \rightsquigarrow$

Arrays in C

- Arrays are addresses: $a[i] \equiv a + i$ (where i is multiplied by the size of a 's elements)
- Multi-dimensional arrays always rectangular, and arranged in *row-major order*:
 - a is declared as `int[10][20]`
 - address of $a[i][j] = a + i*80 + j*4$.

Arrays in C (cont.)

- a is declared as `int[10][20][30]`
- address of `a[i][j][k]` = $a + i * 2400 + j * 120 + k * 4$
- Rule is: address of $e_1[e_2]$ = address of $e_1 + (e_2 * (\text{size of elements of } e_1))$

Array assignment in C

- With arrays, left-hand sides of assignment statements can be complex expressions.
- A compilation scheme like this one makes no sense:

$$\begin{aligned} e_1[e_2] = e_3, m &\rightsquigarrow il1 \text{ @ } il2 \text{ @ } [\text{MOV } loc1, loc2] \\ e_1[e_2], \text{loc1} &\rightsquigarrow il1 \\ e_3, \text{loc2} &\rightsquigarrow il2 \end{aligned}$$

- Consider $a[i] = a[i]$. Can't evaluate both occurrences of $a[i]$ to the same value!

Array assignment in C (cont.)

- “l-values” vs. “r-values”
 - l-values are values of expressions on left-hand sides of assignments. They are *addresses*.
- Need scheme for calculating l-values.
- Compilation of assignment becomes:

$$e_1 = e_2, m \rightsquigarrow il1 @ il2 @ [\text{MOVIND } loc1, loc2]$$
$$e_1, \text{loc1} \rightsquigarrow_{lval} il1$$
$$e_2, \text{loc2} \rightsquigarrow il2$$

Switch statements

- Switch statements can be compiled two ways:
 - As cascade of if statements.
 - As “jump table” — array of locations of the code for each case; use switch expression as index.
- When should you use one or the other?

Objects

- **Fields: How is inheritance of fields handled in our compiler?**
- **Methods: How is inheritance, and overriding, of methods handled in our compiler?**

V-tables

- There is one place in our code where a name appears in the compiled code: when compiling “`new C()`”.
- Why is it needed? (Hint: not to determine the size of the object to be allocated.)
- How can it be eliminated?

V-tables (cont.)

- Draw a table for each class, listing all the methods belonging to that class (including inherited ones). The order should be from top of the hierarchy to the bottom.

```
class B {  
    void f() {}  
    void g() {}  
}
```

```
class C1 extends B {  
    void h() {}  
}
```

```
class C2 extends B {  
    void g() {}  
}
```

```
class D extends C1 {  
    void i() {}  
    void g() {}  
}
```