CSCI 742 - Compiler Construction

Lecture 31
Code Generation for Control Structures
Instructor: Hossein Hojjat

April 17, 2017
Recap: Code Generation for Expressions

\[
\begin{align*}
[e_1 + e_2] &= \\
&= [e_1] \\
&\quad [e_2] \\
&\quad \text{iadd}
\end{align*}
\]

\[
\begin{align*}
[e_1 \times e_2] &= \\
&= [e_1] \\
&\quad [e_2] \\
&\quad \text{imul}
\end{align*}
\]
Recap: Code Generation for Expressions

Code generation visits AST nodes in post-order

```
iconst_1
iconst_2
iadd
iconst_3
iconst_4
iadd
imul
```
JVM Boolean Type

- Although JVM defines a boolean type, it only provides very limited support for it.
- There are no JVM instructions solely dedicated to operations on boolean values.
- Instead, expressions in Java that operate on boolean values are compiled to use values of int.

Java Virtual Machine Specification
Java SE 8 Edition

- We represent Java boolean false in JVM by the integer 0.
- We represent Java boolean true in JVM by the integer 1.
true, false, variables

- \([true] = \text{iconst}_1\)
- \([false] = \text{iconst}_0\)
- for boolean variable \(b\), for which \(n = \text{slotOf}(b)\)
- \([b] = \text{iload}_n\)
- \([b = e] =
  \begin{align*}
  & [e] \\
  & \text{istore}_n
  \end{align*}\)
Compiling `if` Statement

- Recap: `if<cond>` branches if int comparison with zero succeeds

\[
\begin{align*}
\text{if } (\text{cond}) & \text{ tStmt else eStmt} &= \\
\text{cond} & \\
\text{ifeq(nElse)} & \\
\text{tStmt} & \\
\text{goto(nAfter)} & \\
\text{nElse: } & \text{eStmt} \\
\text{nAfter: } & \\
\end{align*}
\]
Compiling **while** Statement

\[
\begin{align*}
(\neg \textit{cond})? \\
\textit{stmt} \\
(\textit{cond})? \\
\textit{nExit}
\end{align*}
\]

\[
\begin{align*}
\textit{nStart} : &\quad [\textit{cond}] \\
&\quad \text{ifeq}(\textit{nExit}) \\
&\quad [\textit{stmt}] \\
&\quad \text{goto}(\textit{nStart}) \\
\textit{nExit} : &
\end{align*}
\]

**Exercise:** Give a translation with only one jump during loop.
Compiling while Statement

\[
\text{while } (cond) \text{ stmt} =
\]

\[
\begin{align*}
\text{nStart:} & \quad [cond] \\
\text{ifeq(nExit)} & \quad [stmt] \\
\text{goto(nStart)} &
\end{align*}
\]

\[
\text{nExit:}
\]

**Exercise:** Give a translation with only one jump during loop
Compiling while Statement

\[
\begin{align*}
(-cond) & \quad \Rightarrow \quad nStmt \\
(\text{cond})? & \quad \Rightarrow \quad stmt \\
\end{align*}
\]

\[\text{while (cond) stmt} =\]

\[
\begin{aligned}
nStart: & \quad [\text{cond}] \\
& \quad \text{ifeq}(\text{nExit}) \\
& \quad [\text{stmt}] \\
& \quad \text{goto}(\text{nStart}) \\
nExit: & \quad \text{goto}(\text{nStart})
\end{aligned}
\]

\[\text{while (cond) stmt} =\]

\[
\begin{aligned}
\text{goto(nStart)} \\
nStmt: & \quad [\text{stmt}] \\
nStart: & \quad [\text{cond}] \\
& \quad \text{ifneq}(\text{nStmt}) \\
nExit: & \quad \text{goto}(\text{nStart})
\end{aligned}
\]

**Exercise:** Give a translation with only one jump during loop
Example: Code Generation for `while` Loop

```java
static boolean cond(int n) {
    /* ...*/
}
static int work(int n) {
    /* ...*/
}
static void func(int n) {
    while(cond(n)) {
        n = work(n);
    }
}
```

```
0: iload_0
1: invokestatic #2 // cond:(I)Z
4: ifeq 15
7: iload_0
8: invokestatic #3 // work:(I)I
11: istore_0
12: goto 0
15: return
```
Exercise

• Oberon-2 has a LOOP statement that expresses repetitions with exit condition in the middle of the loop
• This generalizes while and do ... while
• Give a translation scheme for the LOOP construct

LOOP
  code1
  EXIT IF cond
  code2
END
Exercise

- Oberon-2 has a **LOOP** statement that expresses repetitions with exit condition in the middle of the loop
- This generalizes **while** and **do ... while**
- Give a translation scheme for the **LOOP** construct

```
LOOP
  code1
  EXIT IF cond
  code2
END
```

```
nStart:  [code1]
         [cond]
         ifndef(nExit)
         [code2]
         goto(nStart)
```

```
nExit:  
```
Bitwise Operations

01001000 & 10101110 = 00001000
10101110 = 11101110

- iand computes the bitwise and of value1 and value2 - (which must be ints)
- The int result replaces value1 and value2 on stack

- ior: dual of iand
Compiling Bitwise Operations

\[
\begin{align*}
[e_1 \& e_2] &= \\
&\begin{bmatrix} e_1 \\ e_2 \end{bmatrix} \\
&\text{Iand}
\end{align*}
\]
Short-circuit Evaluation

- Non-bitwise operators && and || are short-circuit operators in Java
- They only evaluate their second operand if necessary
- Must compile short-circuit operators correctly
- It is not acceptable to emit code that always evaluates both operands of &&, ||

\[
\begin{align*}
[e_1 \&\& e_2] &= \\
[e_1] &\times \\
[e_2] &\times \\
\ldots &
\end{align*}
\]

not allowed to evaluate \(e_2\) if \(e_1\) is false

Also for \((e_1\|e_2)\): if \(e_1\) true, \(e_2\) not evaluated
• What does this program do?

```java
static boolean bigFraction(int x, int y) {
    return ((y == 0) | (x/y > 100));
}
public static void main(String[] args) {
    bigFraction(10, 0);
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
• What does this program do?

```java
static boolean bigFraction(int x, int y) {
    return ((y == 0) || (x/y > 100));
}
public static void main(String[] args) {
    bigFraction(10, 0);
}
```

should be `||`

• Exception in thread "main" java.lang.ArithmeticException: / by zero
What does this program do?

```java
static int iterate() {
    int[] x = new int[10];
    int i = 0;
    int res = 0;
    while ((i < x.length) && (x[i] >= 0)) {
        i = i + 1;
        res = res + 1;
    }
    return res;
}
```
What does this program do?

```java
static int iterate () {
    int[] x = new int[10];
    int i = 0;
    int res = 0;
    while ((i < x.length) && (x[i] >= 0)) {
        i = i + 1;
        res = res + 1;
    }
    return res;
}
```

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
Conditional Expression

c ? t : e means:

1. evaluate c
2. if c is true, then evaluate t and return
3. if c is false, then evaluate e and return

- To compile ||, && transform them into conditional expression

\[(p \&\& q) == (p) ? q : false\]
\[(p || q) == (p) ? true : q\]
Compiling Conditional Expression

- Same as for if statement, even though code for branches will leave values on the stack

\[
[(\text{cond}) \ ? \ t \ : \ e] =
\]

\[
[\text{cond}]
\]

\[
\text{ifeq} (\text{nElse})
\]

\[
[t]
\]

\[
\text{goto} (\text{nAfter})
\]

\text{nElse:} \quad [e]

\text{nAfter:}
int f(boolean c, int x, int y) {
    return (c ? x : y);
}
\[
[(\text{cond}) \ ? \ t \ : \ e] = \\
[\text{cond}] \\
\text{ifeq(nElse)} \\
[t] \\
\text{goto(nAfter)} \\
\text{nElse: } [e] \\
\text{nAfter: }
\]

\[
[p \ \&\& \ q] = \\
[(p) \ ? \ q \ : \ \text{false}] = \\
[p] \\
\text{ifeq(nElse)} \\
[q] \\
\text{goto(nAfter)} \\
\text{nElse: } \text{iconst}_0 \\
\text{nAfter: }
\]
\[
\begin{align*}
[(\text{cond}) \ ? \ t \ : \ e] &= [\text{cond}] \\
& \text{ifeq(nElse)} \\
& [t] \\
& \text{goto(nAfter)} \\
\text{nElse:} & \ [e] \\
\text{nAfter:}
\end{align*}
\]

\[
[p \ || \ q] = [(p) \ ? \ \text{true} \ : \ q] = [p] \\
& \text{ifeq(nElse)} \\
& \text{iconst_1} \\
& \text{goto(nAfter)} \\
\text{nElse:} & \ [q] \\
\text{nAfter:}
\]