CSCI 742 - Compiler Construction

Lecture 29
Code Generation for Expressions
Instructor: Hossein Hojjat

April 10, 2017
Recap: Java Virtual Machine (JVM)

- JVM is a stack machine: evaluation of expressions uses a stack (operand stack)
- Instructions fetch their arguments from the top of the operand stack
- Instructions store their results at the top of the operand stack
• Bytecodes are executed in a **postfix** manner
• Postfix: notation for writing arithmetic expressions in which the operands appear before their operators

**Example.**

• Postfix expression:

\[ 1 \ 2 \ + \]

• Bytecode instructions:

```
iconst_1
iconst_2
iadd
```
Let \( f \) be a binary operation, \( e_1 \) and \( e_2 \) two expressions.

We can denote application \( f(e_1, e_2) \) as follows:

- in prefix notation: \( f \ e_1 \ e_2 \)
- in infix notation: \( e_1 \ f \ e_2 \)
- in postfix notation: \( e_1 \ e_2 \ f \)

Suppose that each operator (like \( f \)) has a known number of arguments. For nested expressions:

- infix requires parentheses in general
- prefix and postfix do not require any parentheses
Expressions in Different Notation

- For infix, assume \( \ast \) binds stronger than \(+\)
- There is no need for priorities or parentheses in the other notations

**prefix**

\[ + \ x \ y \ 
+ \ast \ x \ y \ z \ 
+ \ x \ast \ y \ z \ 
\ast \ x \ + \ y \ z \]

**infix**

\[ x \ + \ y \ 
\ast \ y \ + \ z \ 
x \ + \ y \ast \ z \ 
x \ast \ (y \ + \ z) \]

**postfix**

\[ x \ y \ + \ 
\ast \ y \ + \ z \ 
x \ y \ z \ast \ + \ 
x \ y \ z \ast \ + \ 
\]

- Infix is the only problematic notation and leads to ambiguity
- Why is it used in math? Ambiguity reminds us of algebraic laws:
  - \( x + y \) looks same from left and from right (commutative)
  - \( x + y + z \) parse trees mathematically equivalent (associative)
Exercise

- Convert the following expressions into prefix and postfix

infix: 

- $(x + y) \ast (z - y)$
- $(((x + y) + z) + t)$
Exercise

- Convert the following expressions into prefix and postfix

<table>
<thead>
<tr>
<th>prefix</th>
<th>*  +  x  y  −  z  y</th>
<th>+  +  +  x  y  z  t</th>
</tr>
</thead>
<tbody>
<tr>
<td>infix</td>
<td>(x  +  y)  *  (z  −  y)</td>
<td>(((x  +  y)  +  z)  +  t)</td>
</tr>
<tr>
<td>postfix</td>
<td>x  y  +  z  y  −  *</td>
<td>x  y  +  z  +  t  +</td>
</tr>
</tbody>
</table>
Advantage of postfix expressions:

- we can evaluate postfix expressions easier by using a stack

```java
public int evaluate(Expression expression) {
    Scanner scanner = new Scanner(expression);
    Stack<Integer> operands = new Stack<Integer>();
    while (scanner.hasNext()) {
        if (scanner.hasNextInt()) {
            operands.push(scanner.nextInt());
        } else {
            Integer operand2 = operands.pop();
            Integer operand1 = operands.pop();
            String operator = scanner.next();
            switch (operator) {
                case "+" : operands.push(operand1 + operand2); break;
                case "-" : operands.push(operand1 - operand2); break;
                case "*" : operands.push(operand1 * operand2); break;
                case "/" : operands.push(operand1 / operand2); break;
            }
        }
    }
    return operands.pop();
}
```
• Evaluating Infix Needs Recursion

```java
public int evaluate(Expression e) {
    if (e.isInt())
        return e.intValue();
    else {
        switch (e.toString()) {
            case "+": return evaluate(e.left) + evaluate(e.right);
            case "-": return evaluate(e.left) - evaluate(e.right);
            case "*": return evaluate(e.left) * evaluate(e.right);
            case "/": return evaluate(e.left) / evaluate(e.right);
        }
    }
}
```

• Maximal stack depth in interpreter = expression height
Compiling Expressions

- Evaluating postfix expressions is like running a stack-based virtual machine on compiled code
- Compiling expressions for stack machine is like translating expressions into postfix form

**Infix:** \((x + y) \times z\)

**Postfix:** \(x\ y\ +\ z\times\)

**Bytecode:**

```
  iload_1  x
  iload_2  y
  iadd     +
  iload_3  z
  imul     *
```

![Expression Tree](image)
To evaluate $e_1 \ast e_2$ interpreter

- evaluates $e_1$
- evaluates $e_2$
- combines the result using $\ast$

Compiler for $e_1 \ast e_2$ emits:

- code for $e_1$ that leaves result on the stack, followed by
- code for $e_2$ that leaves result on the stack, followed by
- arithmetic instruction that takes values from the stack and leaves the result on the stack
Local Variables

- For integers use instructions `iload` and `istore`
- Assigning indices (called slots) to local variables using function
  \[ \text{slotOf}: \ VarSymbol \rightarrow \{0, 1, 2, 3, \ldots\} \]
- How to compute the indices?
- Assign them in the order in which they appear in the tree

```java
class Compiler implements Visitor<Tree> {
    ...
    public List<Bytecode> visit(Var n) {
        return List(ILoad(slotOf(n.name)));
    }
    ...
    public List<Bytecode> visit(Assign stat) {
        return visit(stat.rhs).addAll(IStore(slotOf(stat.lhs)));
    }
    ...
}
```
Global Variables and Fields

- `getfield`
  Get the value of an instance field

- `putfield`
  Write the value of an instance field

- `getstatic`
  Get the value of a static field

- `putstatic`
  Write the value of a static field
Global Variables and Fields

• `class` file includes a data structure called the “constant pool”
• Constant pool is a table of symbolic names
  - e.g. class names, field names, methods names
• When a bytecode instruction refers to a field the reference is a number: it represents an index into the constant pool
  
  `getfield #20`

• Instruction indicates the 20th symbolic name in the constant pool
class Factorial {
    int num_aux = 0;
    public int fact(int num) {
        if (num < 1)
            num_aux = 1;
        else
            num_aux =
                num*(this.fact(num-1));
        return num_aux;
    }
}

public int fact(int);

Code:
0: iload_1
1: iconst_1
2: if_icmpge 13
5: aload_0
6: iconst_1
7: putfield #2 // Field num_aux:I
10: goto 26
13: aload_0
14: iload_1
15: aload_0
16: iload_1
17: iconst_1
18: isub
19: invokevirtual #3 // Method fact:(I)I
22: imul
23: putfield #2 // Field num_aux:I
26: aload_0
27: getfield #2 // Field num_aux:I
30: ireturn

aload_0 refers to receiver object (0th argument), since fact is not static
Shorthand Notation for Translation

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

\[
[e_1 \times e_2] =
\begin{align*}
&[e_1] \\
&[e_2] \\
&\text{imul}
\end{align*}
\]
Compiling If Statement

• Assume we use 0/1 for translating conditions
• Recap: if<cond> branches if int comparison with zero succeeds

\[
\text{[if (cond) } t\text{Stmt else } e\text{Stmt]} = \begin{align*}
&[\text{cond}] \\
&\text{Ifeq(nElse)} \\
&t\text{Stmt} \\
&\text{goto(nAfter)} \\
\text{nElse: } [e\text{Stmt}] \\
\text{nAfter:}
\end{align*}
\]

• We will discuss control structures (if, while, ...) in Lecture 31
Array Manipulation

\[ a = \text{reference - “address” arrays} \]

\[ i = \text{int arrays (and some other int-like value types)} \]

Selected array manipulation operations:

- **newarray, anewarray, multianewarray** - allocate an array object and put a reference to it on the stack
- **aaload, iaload** - take: a reference to array and index from stack, load the value from array and push it onto the stack
- **aastore, iastore** - take: a reference to array, an index, a value from stack, store the value into the array index
- **arraylength** - retrieve length of the array

Java arrays store the size of the array and its type, which enables run-time checking of array bounds and object types.