



CSCI 742 - Compiler Construction

Lecture 29

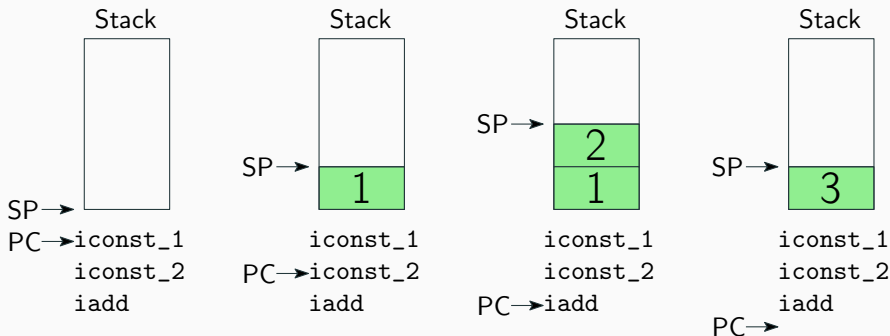
Code Generation for Expressions

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



Prefix, Infix, Postfix

- 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

Prefix, Infix, Postfix

- 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 $*$ binds stronger than $+$
- There is no need for priorities or parentheses in the other notations

prefix	$+ x y$	$+ * x y z$	$+ x * y z$	$* x + y z$
infix	$x + y$	$x * y + z$	$x + y * z$	$x * (y + z)$
postfix	$x y +$	$x y * z +$	$x y z * +$	$x y z + *$

- 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)

- Convert the following expressions into prefix and postfix

infix

$$(x + y) * (z - y)$$

$$(((x + y) + z) + t)$$

Exercise

- Convert the following expressions into prefix and postfix

prefix	$* + x y - z y$	$+ + + x y z t$
infix	$(x + y) * (z - y)$	$((((x + y) + z) + t)$
postfix	$x y + z y - *$	$x y + z + t +$

Postfix Notation

Advantage of postfix expressions:

- we can evaluate postfix expressions easier by using a stack

```
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();
}
```


Infix Notation

- Evaluating Infix Needs Recursion

```
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: $(x + y) * z$

postfix: $x y + z *$

bytecode:

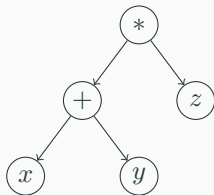
`iload_1` `x`

`iload_2` `y`

`iadd` `+`

`iload_3` `z`

`imul` `*`



Compiling Trees into Bytecodes

To evaluate $e_1 * e_2$ interpreter

- evaluates e_1
- evaluates e_2
- combines the result using $*$

Compiler for $e_1 * e_2$ 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

`slotOf: VarSymbol → {0, 1, 2, 3, ...}`

- How to compute the indices?
- Assign them in the order in which they appear in the tree

```
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

Factorial Example

```
public int fact(int);  
Code:  
0: iload_1  
1: iconst_1  
2: if_icmpge    13  
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;  
  }  
}  
3: aload_0  
4: iconst_1  
5: putfield    #2  // Field num_aux:I  
6: goto       26  
7: aload_0  
8: iload_1  
9: aload_0  
10: iload_1  
11: iconst_1  
12: isub  
13: invokevirtual #3 // Method fact:(I)I  
14: imul  
15: putfield    #2  // Field num_aux:I  
16: aload_0  
17: getfield    #2  // Field num_aux:I  
18: ireturn  
19: aload_0  
20: iconst_1  
21: putfield    #2  // Field num_aux:I  
22: goto       26  
23: aload_0  
24: getfield    #2  // Field num_aux:I  
25: ireturn  
26: aload_0  
27: iconst_1  
28: putfield    #2  // Field num_aux:I  
29: goto       26  
30: ireturn
```

aload_0 refers to receiver object (0th argument), since fact is not static 13

Shorthand Notation for Translation

$$\llbracket e_1 + e_2 \rrbracket =$$
$$\quad \llbracket e_1 \rrbracket$$
$$\quad \llbracket e_2 \rrbracket$$
$$\quad \text{iadd}$$
$$\llbracket e_1 * e_2 \rrbracket =$$
$$\quad \llbracket e_1 \rrbracket$$
$$\quad \llbracket e_2 \rrbracket$$
$$\quad \text{imul}$$

Compiling If Statement

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

```
[[if (cond) tStmt else eStmt]] =  
    [[cond]]  
    Ifeq(nElse)  
    [[tStmt]]  
    goto(nAfter)  
    nElse: [[eStmt]]  
    nAfter:
```

- We will discuss control structures (`if`, `while`, ...) in Lecture 31

Array Manipulation

`a` = reference - “address” arrays

`i` = 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