```
fld a
fld b
fadd
fld c
fld d
fadd
fmul
```

6.6.2 Converting Arithmetic Expressions to Postfix Notation

For simple expressions, those involving two operands and a single expression, the translation from infix to postfix notation is trivial: simply move the operator from the infix position to the postfix position (that is, move the operator from between the operands to after the second operand). For example, `5 + 6` becomes `5 6 +`. Other than separating your operands so you don't confuse them (that is, is it 5 and 6 or 56?), converting simple infix expressions into postfix notation is straightforward.

For complex expressions, the idea is to convert the simple subexpressions into postfix notation and then treat each converted subexpression as a single operand in the remaining expression. The following discussion surrounds completed conversions with square brackets so it is easy to see which text needs to be treated as a single operand in the conversion.

As for integer expression conversion, the best place to start is in the innermost parenthetical subexpression and then work your way outward, considering precedence, associativity, and other parenthetical subexpressions. As a concrete working example, consider the following expression:

```
x = ((y – z) * a) – ( a + b * c ) / 3.14159
```

A possible first translation is to convert the subexpression `(y - z)` into postfix notation:

```
x = ([y z -] * a) - ( a + b * c ) / 3.14159
```

Square brackets surround the converted postfix code just to separate it from the infix code, for readability. Remember, for the purposes of conversion, we will treat the text inside the square brackets as a single operand. Therefore, you would treat `[y z -]` as though it were a single variable name or constant.

The next step is to translate the subexpression `([y z -] * a )` into postfix form. This yields the following:

```
x = [y z - a *] - ( a + b * c ) / 3.14159
```

Next, we work on the parenthetical expression `( a + b * c )`. Because multiplication has higher precedence than addition, we convert `b * c` first:

```
x = [y z - a *] - ( a + [b c *]) / 3.14159
```

20. Convert the following expressions to assembly language (assume all variables are signed 32-bit integers):

```
x = x + y
x = y - z
x = y * z
x = y + z * t
x = (y + z) * t
x = -((x * y) / z)
x = (y == z) && (t != 0)
```

21. Compute the following expressions without using an `imul` or `mul` instruction (assume all variables are signed 32-bit integers):

```
x = x * 2
x = y * 5
x = y * 8
```

22. Compute the following expressions without using a `div` or `idiv` instruction (assume all variables are unsigned 16-bit integers):

```
x = x / 2
x = y / 8
x = z / 10
```

23. Convert the following expressions to assembly language by using the FPU (assume all variables are `real8` floating-point values):

```
x = x + y
x = y - z
x = y * z
x = y + z * t
x = (y + z) * t
x = -((x * y) / z)
```

24. Convert the following expressions to assembly language by using SSE instructions (assume all variables are `real4` floating-point values):

```
x = x + y
x = y - z
x = y * z
x = y + z * t
```

25. Convert the following expressions to assembly language by using FPU instructions; assume `b` is a one-byte Boolean variable and `x`, `y`, and `z` are `real8` floating-point variables:

```
b = x < y
b = x >= y && x < z
```