Previous Contents Next

Chapter 13:
Building a calculator

No reckoning made, but sent to my account
With all my imperfections on my head.
— Shakespeare, *Hamlet*

13.1 Handling operator precedence

It's time to return to the calculator program that was developed in chapter 3, which was capable of evaluating arithmetic expressions like 2+3*4. Expressions were dealt with on a strict left-to-right basis, so that 2+3*4 would evaluate to 20. However, the normal rules of arithmetic tell us that the value of this expression should be 14, since the conventional interpretation involves performing multiplication before addition.

Modifying the calculator to deal with this will require delaying the addition operation until the multiplication has been performed. This is a well-known problem; algorithms to deal with this were first developed in the 1950s and refined in the 1960s. One approach is to use a data structure known as a **stack**. A stack is a collection with specific restrictions on how it can be accessed; the traditional comparison is with a pile of plates. New items can be added to the top of the stack (i.e. you can put more plates on top of the pile) and items can be removed from the top of the stack (i.e. you can remove plates from the top of the pile). The conventional names for these operations are **pushing** an item onto the stack and **popping** an item off the top. You can generally tell if someone is a programmer by asking what the opposite of 'push' is; programmers say 'pop', everyone else says 'pull'! There may be a few extra operations; for example, you may be able to find out how many items the stack contains or inspect the top item without removing it. What you can't do is add or remove items anywhere except at the top of the stack; if you were to try with a stack of plates the result might be a 'stack crash'! A stack is said to have a **last-in first-out** (**LIFO**) organisation: the last item pushed onto the stack is the first one to be popped off.

Stacks are one of the most generally useful data structures around. They crop up in all sorts of situations; for example, the compiler relies on using a stack to keep track of procedure calls. When you call a procedure, your return address (the point in the calling procedure that you want to return to) is pushed onto a stack; returning from a procedure is simply a matter of popping the return address off the top of the stack and going back to the place it specifies. Stacks can also be used for

evaluating arithmetic expressions according to the conventional rules of arithmetic.

The method for doing this requires two stacks, one for operands and one for operators. Whenever you see an operand, you put it on the operand stack; when you see an operator, you compare it with the operator on top of the operator stack. Each operation is given a priority (or precedence); multiplication has a higher priority than addition. If the operator you've just read has a higher priority than the one on top of the stack, you just push it onto the stack. This defers dealing with high-priority operators until you've had a chance to see what comes next. Otherwise, you remove the operator from the top of the stack, remove the top two values from the operand stack, apply the operator to the two operands and push the result onto the operand stack. You then repeat the process until the operator you're considering does have a higher priority than the one on top of the operator stack. In other words, when you see a low-priority operator you first of all deal with any deferred operators on the stack which have the same priority or higher. Finally, you push the operator you're considering onto the operator stack until you see what comes next.

To make this work, you need to prime the operator stack with an operator which has a lower priority than any other. At the end of the expression, operators must be removed one by one from the operator stack together with the top two operands from the operand stack; each operator is applied to its two operands and the result is pushed onto the operand stack. When you reach the low-priority operator on the bottom of the stack, the operand stack will contain a single value which is the result of the expression. Here's what happens if you evaluate 2+3*4+5 using this algorithm:

	Input	Symbol	Operands	Operators	Action
1)	2+3*4+5.			#	(start state)
2)	+3*4+5.	2	2	#	Push 2
3)	3*4+5.	+	2	# +	+ > #; push +
4)	*4+5.	3	2 3	# +	Push 3
5)	4+5.	*	2 3	# + *	* > +; push *
6)	+5.	4	2 3 4	# + *	Push 4
7)	5.	+	2 12	# +	+ < *; apply *
8)	5.	+	14	#	+ = +; apply +
9)	5.	+	14	# +	+ > #; push +
10)	.	5	14 5	# +	Push 5
11)	.		19	#	. < +; apply +

The result is 19. I've used '#' to represent the low-priority operator used to prime the operator stack. Operands and operators are pushed onto their respective stacks until step 7 is reached. Here we've got '+' and the operator stack has '*' on top. So the multiplication operator is removed from the operator stack, the top two items are removed from the operand stack (3 and 4), the multiplication operator is applied to the two operands, and the result (12) is pushed back onto the operand stack. Now we have a '+' on top of the operator stack, so step 8 repeats the process; 2+12 gives 14 which is pushed onto the operand stack. Now the top of the operator stack is '#', which has a lower priority than '+', so the '+' finally gets pushed onto the operator stack in step 9. At step 11 we've reached the full stop which signifies the end of the expression, so the '+' on top of the operator stack is removed, the two operands 14 and 5 are removed from the operand stack, and the result (19) is pushed onto the operand stack. The top operator on the stack is now '#', so the value on top of the operand stack (19) is the final result.
