Chapter 15 Nesting

Section 15.1 What is nesting?

Nesting, in computer programming, is a situation where something is placed within another. We will call these the delimiters of the nesting. Examples of nesting we have seen so far include arithmetic expressions, lists of lists, or function composition (nested function invocations). In this chapter we will see how to nest other things we have just learned about: nested if statements, and nested for statements.

Not only can nesting occur in several places within a structure, a single location can have nests within nests within nests. This is referred to as multiple levels of nesting. Usually there are symbols to mark the beginning and end of the "something in the midst". For arithmetic expressions this would be parentheses ( ), for lists this would be brackets [ ], and for sets this would be \{ \}.

<table>
<thead>
<tr>
<th>Example 15.1 Examples of nesting</th>
<th>Type of nesting</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arithmetic or algebraic expressions</td>
<td>$\frac{3\cdot(x+1)}{y+1} + 3$</td>
<td>The boundaries of the interior expression are delimited by left and right parentheses -- ( and ). This expression has two levels of nesting.</td>
</tr>
<tr>
<td></td>
<td>Nested lists</td>
<td>[[1, 5], [2, 6], [3, 7]]</td>
<td>There are three lists nested inside the bigger list. This makes it a list of lists. There are only two levels of nesting however -- the &quot;5&quot; is within a list which is within another list.</td>
</tr>
<tr>
<td></td>
<td>Nested function invocations (also known as function composition)</td>
<td>$\sin(\cos(\pi + 5))$</td>
<td>Figuring out the value of this on some calculators would require an &quot;inside out&quot; evaluation -- first approximate the value of $\pi + 5$, then calculate cosine of that number, then calculate sin of that number. This has four levels of</td>
</tr>
</tbody>
</table>

$\sqrt{\log_{10}(|\tan(2.0)|)}$
plot(diff(sin(x/sqrt(x)), x), x = 1..trunc(ln(2009))

nesting, since there are four functions (sqrt, log base 10, absolute value, and tangent) being composed here.

While some of these functions are Maple computational functions rather than math functions, it's still functional composition. Note that some of these function have multiple arguments, and that there are at least two different places within plot where function invocations are being nested. The x within the sqrt is nested 4 levels deep, while the 2009 is nested three levels deep.

Section 15.2 The complexity of nested expressions

Nested situations can be more complicated to read. Typically the outer situation begins, then in the middle of it one has to read the inner situation, understand that, then go back to reading and understanding the outer situation. If there are more than two levels, one gets interruptions within interruptions. Typically nesting to a small number of levels is considered acceptable, but too many levels may be detrimental to understanding. How many levels is "too many" depends on the experience and sophistication of the audience. Doubly or triply nested expressions are fairly common. They occur a lot in programming (not just Maple programming, but that of most programming languages) because they are a succinct way of specifying the order of actions in many common and useful situations.

When entering heavily nested expressions, a common error is to have too few closing delimiters—right parentheses ) for functions or arithmetic expressions, right brackets ] for lists. This typically causes an error message and parks the Maple cursor at the location where the lack of a closing symbol was noticed. Often (but not always) the problem can be fixed by inserting the missing marker symbol where the cursor is.

Other kinds of errors that can occur involve having too many or too few opening markers -- left parentheses ( or left brackets [. Another kind of mistake is to put the wrong kind of bracket in, which causes a nonsensical order of symbols.

Example 15.2.1 Working with the complexity of nested expressions
Example | Complexity
--- | ---
$3 + 5 \cdot \left( \frac{(1+x)}{\left(1 + \frac{1}{(1+3)}\right)} \right)$ | Missing a parentheses highlights the first place that Maple noticed that something was not right. In this case, it will draw red dashed lines around the "(1+3)" which turns out to be the place where there's a missing closing parentheses.

$[[1,3],[1,5],[3,6]]$ | In these four examples, we have omitted a left or right "square bracket", or put too many in. We get the same message in each case, but there's something different missing each time. This illustrates the limitations of the helpfulness of the error message. It's because Maple can't really figure out what the exact problem is, so it just highlights the entire expression. The only effective way to deal with this is to have a clear notion of what you want (in this case, a list of lists, where every inner list has the same number of elements), and to find out where the input deviates from the model. The count/balance checking mentioned in this section would be helpful here. For example, with the first expression, you'd discover that the count for the finished expression ended at +1, meaning that there was one more [ than ]. That might be enough to cause you to notice that there's no final ]

$\sin(\cos(\pi + x))$ | Missing a delimiter with nested function invocations doesn't always produce the "unable to match delimiters" message. In general, seeing an error message about "unable to match" or "missing" should be regarded as symptoms of a problem, not as providing details as to what exactly as wrong. Given the frequency of "wrong/missing" delimiter
Error, unable to match delimiters

plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

Forgot inside )
Error, missing operation
plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

Forgot inside )
Error, unable to match delimiters
plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

( is ) by mistake, and forgot to put an inside )
Warning, unable to evaluate the function to numeric values in the region; see the plotting command's help page to ensure the calling sequence is correct

PLOT(...), x = 1 .. 7 (1.2.1)

plot(sin(cos(sqrt(x))), x = 1 .. trunc(log(2009)))

This is the one that was intended, just in case you were wondering.
Count/balance checking An easy way to check that the nesting parentheses are balanced is to start from zero. Every time you see an opening symbol (, add one, every time you see a closing symbol subtract one. At the end of the expression that count should be zero again. There is definitely a mistake if the count ever becomes negative -- more closing )s then opening (s, or if you reach the end of the expression and the count is still positive -- not enough closing )s.

Figure 15.2.1 Counting balanced parentheses

| [ [ 1,3], [1,5], [[3,6]]; | Starting from 0, each time we encounter an opening -- left -- bracket we increment the count. Each time we encounter a closing -- right -- bracket we decrement the count. If the count is 0 at the end of the expression, it indicates that the number of opening and closing brackets is balanced. Here the count indicates that there is one more opening bracket than closing bracket. The fact that the count is always above 1 strongly suggests that the problem is that there is missing closing parentheses is the one that should match the opening [.

| 1 2 1 2 1 23 21| This count indicates that there are two fewer closing )s than there are opening (s. Finding which ones are missing requires thinking about what you're trying to express. The plot expression $\sin(\cos(\sqrt{x}))$ is supposed to be nested within as the first argument to plot. Therefore, the balance count after it should be 1 (the value when we started that subexpression) rather than 2. This leads us to
Evaluation of nested expressions occurs "inside out", with binding occurring "outside in"

Another part of the difficulty of understanding nested expressions comes when you try to figure out the value of the expression. The only way things make sense is to start to evaluate the innermost expression first and then work outwards.

Example 15.2.2 Evaluation of nested expression

### Evaluating the innermost expression

\[
\sin \left( \cos \left( \frac{\pi}{2} \right) - \pi \right)
\]

Evaluating this expression goes from innermost.

\[
0
\]  \hspace{1cm} (1.2.2)

### Evaluating the intermediate expression

\[
tl := \cos \left( \frac{\pi}{2} \right)
\]

We take the result of evaluating the inner expression is 0. The next step is to evaluate the expression of the next level outwards: 0 - \(\pi\).

\[
t2 := tl - \pi
\]

\[
-\pi
\]  \hspace{1cm} (1.2.4)

### Evaluating the outermost expression

\[
\sin(t2)
\]

Evaluating the outermost expression requires using the result of evaluating the inner layers. The \(tl := ..; t2 := ....; \sin(t2)\) sequence of operations produces the same result as single nested expression. Which one you choose to enter depends on which you find more intelligible.

\[
0
\]  \hspace{1cm} (1.2.5)

### Evaluating more complex expressions

\[
\sin \left( \cos \left( \frac{\pi}{2} \right) - \frac{\pi}{2} \right) + \log \left( \tan \left( \frac{\pi}{4} \right) \right)
\]

Evaluating this expression requires evaluating two nested expressions separately and then adding the results together. The first nested

\[
(1.2.6)
\]
expression evaluates to -1, as you can verify by doing the same thing as in the previous line. The second part of the expression evaluates to 0, so the overall result is -1.

With programming functions such as `seq`, or `sum`, evaluating the innermost expression may result not in a number, but in an expression involving symbols that are mentioned in the operations that enclose the innermost nest. These expressions are used for the outer expressions.

<table>
<thead>
<tr>
<th>Example 15.2.3 Evaluation of nested expressions with outer bindings</th>
</tr>
</thead>
</table>
| \[
\sum_{i=1}^{3} \sum_{j=1}^{2} i \cdot (j + 1)^2
\]  
\[
Evaluating the innermost expression is  
\sum_{j=1}^{2} i \cdot (j + 1)^2 = 13i  
\text{. Once we have a value for that, we can evaluate }  
\sum_{i=1}^{3} 13i = 78.
\]
| \[
\sum_{n=5}^{n} (n - 5 + 3 (n + 1)^2 + 2 (n + 1)^3)
\]  
\[
Evaluating the innermost expression produces  
\sum_{j=1}^{n} i \cdot (j + 1)^2  
\]
\[
= \frac{1}{6} i (n + 1) + \frac{1}{2} i (n + 1)^2 + \frac{1}{3} i (n + 1)^3 - i
\]
\[
\text{. Evaluating the outer expression is  }  
\sum_{i=1}^{3} \left( \frac{1}{6} i (n + 1) + \frac{1}{2} i (n + 1)^2 + \frac{1}{3} i (n + 1)^3 - i \right)  
\]
\[
= n - 5 + 3 (n + 1)^2 + 2 (n + 1)^3
\]
| \[
\text{seq}\left( i \binom{10}{i}, i = 1 \ldots 10 \right)
\]  
\[
[10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
\]
| Evaluating the innermost expression \( \binom{10}{i} = \text{binomial}(10, i) \). This symbolic result is fed into the next layer, becoming the first argument to the \text{seq} function. The \text{seq} function produces a sequence of values that result from evaluating the first argument at \( i = 1, i = 2, \ldots i = 10 \), producing \text{binomial}(10, 1), \text{binomial}(10, 2), \ldots \text{binomial}(10, 10). \text{Since all the arguments to the binomials are numeric, Maple can evaluate the symbolic expression to produce the numeric results 10, 45, \ldots 1. The enclosing [...] s turn this sequence into the}
list.

\[
[\text{seq}(\text{op}(\text{seq}([i, \cdot j], j = 1 .. 2)), i = 1 .. 3)] =
[[1, 2], [1, 4], [2, 2], [2, 4], [3, 2], [3, 4]]
\]

Evaluating the inner expression produces the list \([\text{seq}([i, \cdot j], j = 1 .. 2)] = [[i, 2], [i, 4]]. Evaluating the next layer produces \(\text{op}([[i, 2], [i, 4]]) = [i, 2], [i, 4]. Evaluating the next layer produces \([\text{seq}(\text{op}([[i, 2], [i, 4]]), i = 1 .. 3)]

\[
[[1, 2], [1, 4], [2, 2], [2, 4], [3, 2], [3, 4]]
\]

Thus the overall result is a list of lists.

While this works as a way of producing a list of lists, there is no doubt that the process is intricate and hard for Maple novices to understand. In section 15.4 we present a different way of doing this which is to some simpler to grasp.

### Section 15.3 Working through cases and subcases with nested ifs.

As explained in Chapter 14, if statements come in \textbf{if... then...then...else, and if...then...elif...} forms.

<table>
<thead>
<tr>
<th>Table 15.3.1 General forms of an if statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>if \textit{condition} then \textit{one or more statements} else* \textit{one or more statements} end if;</td>
</tr>
<tr>
<td>if \textit{condition1} then \textit{one or more statements} elif \textit{condition2} then \textit{one or more statements} elif ... then ... elif ... then .... else* \textit{one or more statements} end if;</td>
</tr>
</tbody>
</table>

*\textit{else} portion of the statement is optional*

The rules of Maple (and most other programming languages) allow the "one or more statements" to be any variety of statement. While most of the time these statements are assignment statements (lines that use the := operation), they could be any other kind of statement, such as another if. This would be a situation with \textit{nested if statements}. Recall that if statements begin with \textbf{if} and finish with \textbf{end if}, so if you see a second \textbf{if} before the first \textbf{if} ends, then you must be seeing nested if statements.
The evaluation of nested if statements has an important difference from that of the evaluation of nested expressions. Nested expressions are evaluated in the style of mathematical functions -- "inside out". Nested if statements proceeds in the style of programming -- top to bottom.

### Example 15.3.1 Nested if statements

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
</table>
| if `condition1` then  
  if `condition2` then  
  one or more statements A  
  else  
  one or more statements B  
  end if;  
elif `condition3` then  
  if `condition 4` then  
  one or more statements C  
  else  
  one or more statements D  
  end if;  
end if; | This statement is processed in top to bottom order. First, `condition1` is determined to be true or false. If it is true, then `condition2` is evaluated. If it is true (which means that both `condition1` and `condition2` have both been found to be true, then statements A are executed.  
If `condition1` is true and `condition2` is false, then statements B are executed.  
If `condition1` was found to be false, then `condition3` is evaluated. If it is true, then processing turns to the next statement, which causes `condition4` to be evaluated.  
If that is found to be true (which means that `condition1` was found to be false, `condition3` to be true and `condition4` also found to be true, then statements C are executed. Otherwise (which means that `condition1` was false, `condition3` was true, and `condition4` was false, then statements D are executed.  
If `condition1` and `condition3` are both found to be false, then none of the interior statements are executed. |
| This script takes a list of course marks and counts how many are As, Bs, Cs, or below C. It also prints out for each mark, what letter grade it corresponds to.  
We use four variables to count how many As, Bs, Cs, and subCs. We might have used a table with four entries instead to do this. |
#list of marks
marks := [2.4, 3.9, 1.1, 3.2, 2.8, 2.9]:

#initialize counters
numAs := 0;
numBs := 0;
numCs := 0;
numSubCs := 0:

#initialize results table
results := table():

#Process each grade
for i from 1 to nops(marks) do
    grade := marks[i];
    if grade>=3.6
        then
            numAs := numAs+1;
            if grade >=3.8
                then results[i] := "A";
                else results[i] := "A-";
                end if;
        end if;
    elif grade >=2.6
        then
            numBs := numBs +1;
            if grade >=3.3
                then results[i] := "B+";
                elif grade >=2.8
                    then results[i] := "B";
                    else results[i] := "B-";
                    end if;
            end if;
        elif grade >= 1.6
            then
                numCs := numCs + 1;
                if grade >2.3
                    then results[i] := "C+";
                    elif grade >1.8
                        then results[i] := "C-";
                        else results[i] := "C-";
                        end if;
                end if;
            end if;
        end if;
    end if;
end do;

We store the letter grades in a table and will eventually convert it to a list.

The "result[i] := "A-";" statement is executed only if it is known that grade >= 3.6 is true and grade >=3.8 is not true.

The "result[i] := "B+";" statement is executed only if it is known that grade >= 3.6 is not true and grade >=3.3 is true. That is, we know that 3.3 ≤ grade < 3.6. There are similar facts that justify the assignment of "B" and "B-".
Contrast this with the processing of grades that occurred using piecewise expressions in Section 5.8. Using a programming script of several statements rather than a single function is more attractive here because we are doing several things at once -- counting letter grades, as well as printing individual messages for each grade.

We will begin to introduce situations where it makes sense to have nested if statements. After becoming familiar with the concept and patterns of its use, you should realize situations where you can write some yourself.
### Section 15.4 When an action within a *while* or *for* is described through repetition: nested loops

*for* and *while* loops can be nested. Loops begin with *for* or *while* and end with *end do* or *od*. Thus if you see a *for* and then a *while* or *for* before the first one ends with an *end do*, then you must be seeing nested for/while statements.

The evaluation of nested loops proceeds in the top-to-bottom fashion of all programming statements. However, since loops "go back and repeat", it takes some practice to keep straight the order of execution when loops are nested. The general rule of operation is: execution starts at the top of the nested loops and continues forward until the end of the innermost loop is reached. At that point, the statements in the innermost loop are repeated until the loop control decides that the innermost repetition is repeated. Then execution proceeds into the statements of the next layer. If the next layer is also a loop, then that layer is repeated, which may involve re-execution (and re-repetition) of the innermost loop again.

### Example 15.4.1 Nested for loops to just do printing

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| for i from 1 to 3 do  
  for j from 2 to i+1 do  
    print (i, j);  
    end do;  
  print("end inner loop"); 
end do; | This nested loop doesn't do anything useful, but it does illustrate the order of execution of nested loops. Note that *j*, the inner index, goes through a complete cycle of repetition before the value of *i* changes. *j* then goes through another cycle of repetition. |

1, 2  
"end inner loop"  
2, 2  
2, 3  
"end inner loop"  
3, 2  
3, 3  
3, 4  
"end inner loop"
The classic example of a nested for loop is a script that prints out the items in a list of lists, in rows and columns.

```
L := [ [1.3, 1.4, 9.2], [1.9, 2.23, 11.74], [1, 2, 3], [-3.2, 1.7, 10.3] ]:
numRows := 4:
numCols := 3:

for i from 1 to numRows do
    printf("Row %d:", i);
    for j from 1 to numCols do
        printf("%6.2f", L[i][j]);
    end do;
    printf("\134134134n"); # end the row of output
end do;
```

Row 1:  1.30  1.40  9.20134134134nRow 2:  1.90  2.23 11.74134134134nRow 3:  1.00  2.00 3.00134134134nRow 4: -3.20  1.70 10.30134134134n

This script prints out a pattern of asterisks. The inner loop prints out \( i \) asterisks, all on the same line. The outer loop specifies that after the inner loop finishes, the line should be ended, so that any subsequent printing occurs on the next line.

The overall result is that when \( i = 1 \) the innermost loop executes only once and it
```c
numRows := 4;
for i from 1 to numRows do
    for j from 1 to i do
        printf("*");
    end do;
    printf("
");
end do;
```

prints 1 asterisk (when i=1). When i=2, the innermost loop executes the `printf("*")`; twice, so two asterisks are printed. This continues in a similar way for i=3 and i=4.

We have shown an example of a nested loop with two layers, but it is legal to construct nestings with three or even more layers. Such loops may occur when doing design or simulation of a sophisticated mathematical model with several variables.

### Example 15.4.2 Nested loops that run a probability experiment

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>We run multiple experiments and compute an average result over <code>numTrials</code> experiments.</td>
<td></td>
</tr>
<tr>
<td>The outer loop repeats experiments.</td>
<td></td>
</tr>
<tr>
<td>For each experiment, we count how many rolls of two dice it takes before a &quot;2&quot; shows up (which comes from rolling a &quot;1&quot; from each die).</td>
<td></td>
</tr>
<tr>
<td>The inner loop, controlled by <code>while</code>, runs as long as it takes until a 2 shows up. Each time a 2 does not occur, we add one onto the count. When we roll a 2, the experiment is over and we print out the result for it.</td>
<td></td>
</tr>
</tbody>
</table>
total := 0:
rollDie := rand(1..6):
numTrials := 1000:
for i from 1 to numTrials do
    count := 1;
    while ((rollDie() + rollDie()) <> 2) do
        count := count +1;
    end do;
    if count=1 then
        plural := "";
    else
        plural := "s";
    end if;
    # printf("Took %d roll%s for snake eyes.\134134n", count, plural);
    #
    total := total + count;
end do:
printf("Average number of rolls to make snake eyes = %5.2f", evalf(total/numTrials));

Average number of rolls to make snake eyes = 5.98

Along the way we print out the result of each experiment. At the end we print out the average number of rolls it took to get snake eyes. Note that rather than having two printf statements, we just compute whether we should add an "s" to "roll" and print out the result. Whether this is less work is a matter of taste. Some programmers would just be ungrammatical sometimes and use "rolls" even if the count was 1..

Section 15.5 For the curious: Nesting in other languages

Nested parentheses (and the problems of having a typo created an unbalanced number of left and right parentheses) are found in almost all conventional programming languages, -- since function
composition is often used to express computation, most programming languages support it. Some families of languages (e.g. LISP and Scheme, originally developed at MIT to support artificial intelligence) express all computation through functional composition, even actions such as assignment or loops. Such languages typically benefit from extra features in software editors to help programmers balance parentheses or notice unbalanced sets of parentheses.

Figure 15.4.1 A Scheme program to separate a list of numbers into two and its result.

(let loop
  ((numbers '(3 -2 1 6 -5))
   (nonneg '())
   (neg '()))
  (cond ((null? numbers)
          (list nonneg neg))
       ((>= (car numbers) 0)
        (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
       (else
        (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg))))
  ((6 1 3) (-5 -2))

This procedure initializes numbers to be bound to the list [3, -2, 1, 6, 5] and nonneg and neg. to the empty list []

It then repeatedly loops until numbers is the empty list, deciding of numbers[1] is nonnegative. If it is, to adds that number onto the list "nonneg". Otherwise, it adds it to the list "neg". "loop" in this example is not a Scheme instruction (as "cond", "null?", "cdr", or "cons" are), but just the name the programmer is giving to the function.

The result of running this computation is a list of lists. The first item is a list of non-negative numbers, and the second the list of negative numbers.

Even though the syntax is weird, and the sequence of execution is something called "recursive programming" that we have not yet seen in Maple, we can still use some of the terminology we have learned from studying Maple to talk about what the program is producing and how it decides what to do.

Nested if's, and for's occur as a common pattern in most other conventional programming languages.

Figure 15.4.2 A Python nested for loop that prints out a little multiplication table

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; for multiplier in range(2,5):</td>
<td>Python syntax comes from the</td>
</tr>
</tbody>
</table>
... for j in range (1,5):
...    print "%d x %d = %d" % (j, multiplier, j*multiplier)
...    print "-------------"
...
1 x 2 = 2
2 x 2 = 4
3 x 2 = 6
4 x 2 = 8
-------------
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
-------------
1 x 4 = 4
2 x 4 = 8
3 x 4 = 12
4 x 4 = 16
-------------

same lineage. The range of index variables are specified by "range(2,5)" instead of "from 2 to 5". Python requires loop bodies to be indented. Because of the required indentation Python does not need a symbol like "end do" to signal the end of the loop. The loop ends where there is less indentation.

There are "printf-ish" kinds of things going on in the output with the symbol % separating the format string from the items to be printed.

It should be straightforward for you to translate this idea into a Maple script to do a multiplication table.

Section 15.Z Chapter summary

Nested expressions express function composition. Evaluation proceeds "inside out".

<table>
<thead>
<tr>
<th>Example 15.1 Examples of nesting</th>
<th>Type of nesting</th>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arithmetic or algebraic expressions</td>
<td>$\frac{3 \cdot (x + 1)}{y + 1} + 3$</td>
<td>The boundaries of the interior expression are delimited by left and right parentheses -- ( and ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$3 \cdot \left( 5 + \frac{x}{(1 + x)^2} \right) + x^2$</td>
<td>This expression has two levels of nesting.</td>
</tr>
<tr>
<td></td>
<td>Nested lists</td>
<td>[[1, 5], [2, 6], [3, 7]]</td>
<td>There are three lists nested inside the bigger list. This makes it a list of lists There are only two levels of nesting however -- the &quot;5&quot; is within a list which is within another list.</td>
</tr>
<tr>
<td>Nested function invocations (also known as <em>function composition</em>)</td>
<td>( \sin(\cos(\pi + 5)) )</td>
<td>Figuring out the value of this on some calculators would require an &quot;inside out&quot; evaluation -- first approximate the value of ( \pi + 5 ), then calculate cosine of that number, then calculate sin of that number. This has four levels of nesting, since there are four functions (sqrt, log base 10, absolute value, and tangent) being composed here. While some of these functions are Maple computational functions rather than math functions, it's still functional composition. Note that some of these function have multiple arguments, and that there are at least two different places within <em>plot</em> where function invocations are being nested. The ( x ) within the sqrt is nested 4 levels deep, which the 2009 is nested three levels deep.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>( \sqrt{\log_{10}(</td>
<td>\tan(2.0)</td>
<td>)} )</td>
<td></td>
</tr>
<tr>
<td>( \text{plot}\left(\text{diff}\left(\sin\left(\frac{x}{\sqrt{x}}\right), x\right), x = 1 \ldots \text{trunc}(\ln(2009))\right) )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unbalanced parentheses or brackets can be a common source of error in entering nested expressions. You can count balanced pairs of parentheses or brackets to detect if this is going on within a problem expression.

**Count/balance checking** An easy way to check that the nesting parentheses are balanced is to start from zero. Every time you see an opening symbol (, add one, every time you see a closing symbol subtract one. At the end of the expression that count should be zero again. There is definitely a mistake if the count ever becomes negative -- more closing )s then opening (s, or if you reach the end of the expression and the count is still positive -- not enough closing )s.

**Figure 15.2.1 Counting balanced parentheses**

| Starting from 0, each time we encounter an | |
| --- | |
opening -- left -- bracket we increment the count. Each time we encounter a closing -- right -- bracket we decrement the count. If the count is 0 at the end of the expression, it indicates that the number of opening and closing brackets is balanced. Here the count indicates that there is one more opening bracket than closing bracket. The fact that the count is always above 1 strongly suggests that the problem is that there is missing closing parentheses is the one that should match the opening [.

This count indicates that there are two fewer closing )s than there are opening (s. Finding which ones are missing requires thinking about what you're trying to express. The plot expression \( \sin \left( \cos \left( \sqrt{x} \right) \right) \) is supposed to be nested within as the first argument to \( \text{plot} \). Therefore, the balance count after it should be 1 (the value when we started that subexpression) rather than 2. This leads us to notice that we didn't close the \( \sin( \ldots \) ....

Inserting that parentheses results in an improvement but it evidently is not the only mistake. Since the final count is 1 instead of 0, this suggests that the missing closing parentheses is at the end. Inspection of the expression shows that the opening \( \text{plot}( \) does not have a closing.

Putting this at the end results in a meaningful and balanced expression.

Evaluation of nested expressions proceeds "inside out".

From Examples 15.2.2 and 15.2.3 Evaluation of nested expression

\[
\sin \left( \cos \left( \frac{\pi}{2} - \frac{\pi}{2} \right) + \log \left( \tan \left( \frac{\pi}{4} \right) \right) \right) - 1
\]  \hspace{1cm} \text{(1.6.1)}

Evaluating this expression requires evaluating two nested expressions separately and then adding the results together. The first nested expression evaluates to -1, as you can verify.
by doing the same thing as in the previous line. The second part of the expression evaluates to 0, so the overall result is -1.

\[
\sum_{j=1}^{2} \sum_{i=1}^{3} i \cdot (j + 1)^2
\]

Error, (in sum) summation variable previously assigned, second argument evaluates to 5 = 1 .. 2

Evaluating the innermost expression is

\[
\sum_{j=1}^{2} i \cdot (j + 1)^2 = \text{Error, (in sum) summation variable previously assigned, second argument evaluates to } 5 = 1 .. 2.
\]

Once we have a value for that, we can evaluate \(\sum_{i=1}^{3} 13 \cdot i = \text{Error, (in sum) summation variable previously assigned, second argument evaluates to } 1001 = 1 .. 3\).

\[
[\text{seq}(\text{op}([\text{seq}([i, 2 \cdot j], j = 1 .. 2)], i = 1 .. 3)), i = 1 .. 3) = [\{1, 2\}, [1, 4], [2, 2], [2, 4], [3, 2], [3, 4]]
\]

Evaluating the inner expression produces the list \([\text{seq}([i, 2 \cdot j], j = 1 .. 2)] = [[1001, 2], [1001, 4]]\). Evaluating the next layer produces \(\text{op}([[i, 2], [i, 4]]) = [1001, 2], [1001, 4]\). Evaluating the next layer produces

\[
[\text{seq}(\text{op}([[i, 2], [i, 4]], i = 1 .. 3))] = [[1, 2], [1, 4], [2, 2], [2, 4], [3, 2], [3, 4]].
\]

Thus the overall result is a list of lists.

While this works as a way of producing a list of lists, there is no doubt that the process is intricate and hard for Maple novices to understand. In section 15.4 we present a different way of doing this which is to some simpler to grasp.

Evaluation of nested if proceeds "top down". Statements that are executed within nested if statements are done in the context of several conditions being checked.

**From Example 15.3.1 Nested if statements**

This script takes a list of course marks and counts how many are As, Bs, Cs, or below C. It also prints out for each mark, what letter grade it corresponds to.
#list of marks
marks := [2.4, 3.9, 1.1, 3.2, 2.8, 2.9]:

#initialize counters
numAs := 0;
numBs := 0;
numCs := 0;
numSubCs := 0:

#initialize results table
results := table():

#Process each grade
for i from 1 to nops(marks) do
    grade := marks[i];
    if grade>=3.6
        then
            numAs := numAs+1;
            if grade >=3.8
                then results[i] := "A";
                else results[i] := "A-";
            end if;
    end if;
    elif grade >=2.6
        then
            numBs := numBs +1;
            if grade >=3.3
                then results[i] := "B+";
                elif grade >=2.8
                    then results[i] := "B";
                    else results[i] := "B-";
                end if;
    end if;
    elif grade >= 1.6
        then
            numCs := numCs + 1;
            if grade >=2.3
                then results[i] := "C+";
                elif grade >=1.8
                    then results[i] := "C-";
                end if;
        end if;
    end if;
end do;

We use four variables to count how many As, Bs, Cs, and subCs. We might have used a table with four entries instead to do this.

We store the letter grades in a table and will eventually convert it to a list.

The "result[i] := "A-"; statement is executed only if it is known that grade >=3.6 is true and grade >=3.8 is not true.

The "result[i] := "B+"; statement is executed only if it is known that grade >=3.6 is not true and grade >=3.3 is true. That is, we know that 3.3 ≤ grade < 3.6. There are similar facts that justify the assignment of "B" and "B-".
Contrast this with the processing of grades that occurred using piecewise expressions in Section 5.8. Using a programming script of several statements rather than a single function is more attractive here because we are doing several things at once -- counting letter grades, as well as printing individual messages for each grade.

Evaluation of nested for/while loops proceeds "top down". Statements within an outer loop are repeated. Nesting a for/while loop within another one causes the repetition of the inner loop to recur for each trip through the outer loop.

**Example 15.4.2 Nested loops that run a probability experiment**

<table>
<thead>
<tr>
<th>Example</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>We run multiple experiments and compute an average result over numTrials experiments. The outer loop repeats experiments. For each experiment, we count how many rolls of two dice it takes before a &quot;2&quot; shows up (which comes from rolling a &quot;1&quot; from each die).</td>
<td></td>
</tr>
</tbody>
</table>
total := 0:
rollDie := rand(1..6):
umTrials := 10:
for i from 1 to numTrials do
    count := 1;
    while rollDie() + rollDie() <> 2 do
        count := count +1;
    end do;
    if count=1 then
        plural := "";
    else
        plural := "s";
    end if;
    printf("Took %d roll%s for snake eyes.\134134134n", count, plural);
    total := total + count;
end do:
printf("Average number of rolls to make snake eyes = %5.2f", evalf(total/numTrials));

Took 7 rolls for snake eyes.134134134nTook 10 rolls for snake eyes.134134134nTook 18 rolls for snake eyes.134134134nTook 1 roll for snake eyes.134134134nTook 3 rolls for snake eyes.134134134nTook 8 rolls for snake eyes.134134134nTook 1 roll for snake eyes.134134134nTook 1 roll for snake eyes.134134134nTook 2 rolls for snake eyes.134134134n

The inner loop, controlled by **while**, runs as long as it takes until a 2 shows up. Each time a 2 does not occur, we add one onto the count. When we roll a 2, the experiment is over and we print out the result for it.

Along the way we print out the result of each experiment. At the end we print out the average number of rolls it took to get snake eyes. Note that rather than having two printf statements, we just compute whether we should add an "s" to "roll" and print out the result. Whether this is less work is a matter of taste. Some programmers would just be ungrammatical sometimes and use "rolls" even if the count was 1.
<table>
<thead>
<tr>
<th>To make snake eyes</th>
<th>Average number of rolls to make snake eyes = 5.30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Took 2 rolls for snake eyes</td>
<td></td>
</tr>
</tbody>
</table>