13

Command Scripts

You will probably find that your most intensive use of AMPL's command environment occurs during the initial development of a model, when the results are unfamiliar and changes are frequent. When the formulation eventually settles down, you may find yourself typing the same series of commands over and over to solve for different collections of data. To accelerate this process, you can arrange to have AMPL read often-used sequences of commands from files or to repeat command sequences automatically, determining how to proceed and when to stop on the basis of intermediate results.

A *script* is a sequence of commands, captured in a file, to be used and re-used. Scripts can contain any AMPL commands, and may include programming language constructs like `for`, `repeat`, and `if` to repeat statements and perform them conditionally. In effect, these and related commands let you write small programs in the AMPL command language. Another collection of commands permit stepping through a script for observation or debugging. This chapter introduces AMPL command scripts, using formatted printing and sensitivity analysis as examples.

AMPL command scripts are able to work directly with the sets of character strings that are central to the definition of models. A `for` statement can specify commands to be executed once for each member of some set, for example. To support scripts that work with strings, AMPL provides a variety of string functions and operators, whose use is described in the last section of this chapter.

13.1 Running scripts: `include` and `commands`

AMPL provides several commands that cause input to be taken from a file. The command

include *filename*

is replaced by the contents of the named file. An `include` can even appear in the middle of some other statement, and does not require a terminating semicolon.

The `model` and `data` commands that appear in most of our examples are special cases of `include` that put the command interpreter into model or data mode before reading the specified file. By contrast, `include` leaves the mode unchanged. To keep things simple, the examples in this book always assume that `model` reads a file of model declarations, and that `data` reads a file of data values. You may use any of `model`, `data` and `include` to read any file, however; the only difference is the mode that is set when reading starts. Working with a small model, for example, you might find it convenient to store in one file all the model declarations, a `data` command, and all the data statements; either a `model` or an `include` command could read this file to set up both model and data in one operation.

As an illustration, if the file `dietu.run` contains

```
model dietu.mod;
data dietu.dat;
solve;
option display_1col 5;
option display_round 1;
display Buy;
```

then including it will load the model and data, run the problem, and display the optimal values of the variables:

```
ampl: include dietu.run;
MINOS 5.5: optimal solution found.
6 iterations, objective 74.27382022

Buy [*] :=
BEEF  2.0   FISH  2.0    MCH  2.0    SPG  5.3
 CHK 10.0    HAM  2.0    MTL  6.2    TUR  2.0
;
```

When an included file itself contains an `include`, `model` or `data` command, reading of the first file is suspended while the contents of the contained file are included. In this example, the command `include dietu.run` causes the subsequent inclusion of the files `dietu.mod` and `dietu.dat`.

One particularly useful kind of `include` file contains a list of `option` commands that you want to run before any other commands, to modify the default options. You can arrange to include such a file automatically at startup; you can even have AMPL write such a file automatically at the end of a session, so that your option settings will be restored the next time around. Details of this arrangement depend on your operating system; see Sections A.14.1 and A.23.

The statement

`commands` *filename* `;`

is very similar to `include`, but is a true statement that needs a terminating semicolon and can only appear in a context where a statement is legal.

To illustrate `commands`, consider how we might perform a simple sensitivity analysis on the multi-period production problem of Section 4.2. Only 32 hours of production time are available in week 3, compared to 40 hours in the other weeks. Suppose that we want to see how much extra profit could be gained for each extra hour in week 3. We can accomplish this by repeatedly solving, displaying the solution values, and increasing `avail[3]`:

```
ampl: model steelT.mod;
ampl: data steelT.dat;

ampl: solve;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
ampl: display Total_Profit >steelT.sens;
ampl: option display_1col 0;
ampl: option omit_zero_rows 0;
ampl: display Make >steelT.sens;
ampl: display Sell >steelT.sens;
ampl: option display_1col 20;
ampl: option omit_zero_rows 1;
ampl: display Inv >steelT.sens;

ampl: let avail[3] := avail[3] + 5;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 532033
ampl: display Total_Profit >steelT.sens;
ampl: option display_1col 0;
ampl: option omit_zero_rows 0;
ampl: display Make >steelT.sens;
ampl: display Sell >steelT.sens;
ampl: option display_1col 20;
ampl: option omit_zero_rows 1;
ampl: display Inv >steelT.sens;

ampl: let avail[3] := avail[3] + 5;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 549033
ampl:
```

To continue trying values of `avail[3]` in steps of 5 up to say 62, we must complete another four solve cycles in the same way. We can avoid having to type the same commands over and over by creating a new file containing the commands to be repeated:

```
solve;
display Total_Profit >steelT.sens;
option display_1col 0;
option omit_zero_rows 0;
display Make >steelT.sens;
display Sell >steelT.sens;
option display_1col 20;
option omit_zero_rows 1;
display Inv >steelT.sens;
let avail[3] := avail[3] + 5;
```

If we call this file `steelT.sa1`, we can execute all the commands in it by typing the single line `commands steelT.sa1`:

```
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
1 iterations, objective 532033
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
1 iterations, objective 549033
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
2 iterations, objective 565193
ampl:
```

(All output from the `display` command is redirected to the file `steelT.sens`, although we could just as well have made it appear on the screen.)

In this and many other cases, you can substitute `include` for `commands`. In general it is best to use `commands` within command scripts, however, to avoid unexpected interactions with `repeat`, `for` and `if` statements.

13.2 Iterating over a set: the `for` statement

The examples above still require that some command be typed repeatedly. AMPL provides looping commands that can do this work automatically, with various options to determine how long the looping should continue.

We begin with the `for` statement, which executes a statement or collection of statements once for each member of some set. To execute our multi-period production problem sensitivity analysis script four times, for example, we can use a single `for` statement followed by the command that we want to repeat:

```
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: for {1..4} commands steelT.sa1;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
MINOS 5.5: optimal solution found.
1 iterations, objective 532033
MINOS 5.5: optimal solution found.
1 iterations, objective 549033
MINOS 5.5: optimal solution found.
2 iterations, objective 565193
ampl:
```

The expression between `for` and the command can be any AMPL indexing expression.

As an alternative to taking the commands from a separate file, we can write them as the body of a `for` statement, enclosed in braces:

```
model steelT.mod;
data steelT.dat;

for {1..4} {
   solve;
   display Total_Profit >steelT.sens;
   option display_1col 0;
   option omit_zero_rows 0;
   display Make >steelT.sens;
   display Sell >steelT.sens;
   option display_1col 20;
   option omit_zero_rows 1;
   display Inv >steelT.sens;
   let avail[3] := avail[3] + 5;
}
```

If this script is stored in `steelT.sa2`, then the whole iterated sensitivity analysis is carried out by typing

```
ampl: commands steelT.sa2
```

This approach tends to be clearer and easier to work with, particularly as we make the loop more sophisticated. As a first example, consider how we would go about compiling a table of the objective and the dual value on constraint `Time[3]`, for successive values of `avail[3]`. A script for this purpose is shown in Figure 13-1. After the model and data are read, the script provides additional declarations for the table of values:

```
set AVAIL3;
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
```

The set `AVAIL3` will contain all the different values for `avail[3]` that we want to try; for each such value `a`, `avail3_obj[a]` and `avail3_dual[a]` will be the associated objective and dual values. Once these are set up, we assign the set value to `AVAIL3`:

```
let AVAIL3 := avail[3] .. avail[3] + 15 by 5;
```

and then use a `for` loop to iterate over this set:

```
for {a in AVAIL3} {
   let avail[3] := a;
   solve;
   let avail3_obj[a] := Total_Profit;
   let avail3_dual[a] := Time[3].dual;
}
```

We see here that a `for` loop can be over an arbitrary set, and that the index running over the set (`a` in this case) can be used in statements within the loop. After the loop is com-

```
model steelT.mod;
data steelT.dat;
option solver_msg 0;

set AVAIL3;
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
let AVAIL3 := avail[3] .. avail[3] + 15 by 5;

for {a in AVAIL3} {
   let avail[3] := a;
   solve;
   let avail3_obj[a] := Total_Profit;
   let avail3_dual[a] := Time[3].dual;
}
display avail3_obj, avail3_dual;
```

Figure 13-1: Parameter sensitivity script (`steelT.sa3`).

plete, the desired table is produced by displaying `avail3_obj` and `avail3_dual`, as shown at the end of the script in Figure 13-1. If this script is stored in `steelT.sa3`, then the desired results are produced with a single command:

```
ampl: commands steelT.sa3;
:  avail3_obj avail3_dual :=
32    515033      3400
37    532033      3400
42    549033      3400
47    565193      2980
;
```

In this example we have suppressed the messages from the solver, by including the command `option solver_msg 0` in the script.

AMPL's `for` loops are also convenient for generating formatted tables. Suppose that after solving the multi-period production problem, we want to display sales both in tons and as a percentage of the market limit. We could use a `display` command to produce a table like this:

```
ampl: display {t in 1..T, p in PROD}
ampl?    (Sell[p,t], 100*Sell[p,t]/market[p,t]);
:        Sell[p,t] 100*Sell[p,t]/market[p,t]    :=
1 bands    6000           100
1 coils     307             7.675
2 bands    6000           100
2 coils    2500           100
3 bands    1400            35
3 coils    3500           100
4 bands    2000            30.7692
4 coils    4200           100
;
```

By writing a script that uses the `printf` command (A.16), we can create a much more effective table:

```
ampl: commands steelT.tab1;
SALES          bands            coils
week 1     6000  100.0%       307    7.7%
week 2     6000  100.0%      2500  100.0%
week 3     1399   35.0%      3500  100.0%
week 4     1999   30.8%      4200  100.0%
```

The script to write this table can be as short as two `printf` commands:

```
printf "\n%s%14s%17s\n", "SALES", "bands", "coils";
printf {t in 1..T}: "week %d%9d%7.1f%%%9d%7.1f%%\n", t,
   Sell["bands",t], 100*Sell["bands",t]/market["bands",t],
   Sell["coils",t], 100*Sell["coils",t]/market["coils",t];
```

This approach is undesirably restrictive, however, because it assumes that there will always be two products and that they will always be named `coils` and `bands`. In fact the `printf` statement cannot write a table in which both the number of rows and the number of columns depend on the data, because the number of entries in its format string is always fixed.

A more general script for generating the table is shown in Figure 13-2. Each pass through the ‘‘outer’’ loop over `{1..T}` generates one row of the table. Within each pass, an ‘‘inner’’ loop over `PROD` generates the row’s product entries. There are more `printf` statements than in the previous example, but they are shorter and simpler. We use several statements to write the contents of each line; `printf` does not begin a new line except where a newline (`\n`) appears in its format string.

Loops can be nested to any depth, and may be iterated over any set that can be represented by an AMPL set expression. There is one pass through the loop for every member of the set, and if the set is ordered — any set of numbers like `1..T`, or a set declared `ordered` or `circular` — the order of the passes is determined by the ordering of the set. If the set is unordered (like `PROD`) then AMPL chooses the order of the passes, but

```
printf "\nSALES";
printf {p in PROD}: "%14s   ", p;
printf "\n";
for {t in 1..T} {
   printf "week %d", t;
   for {p in PROD} {
      printf "%9d", Sell[p,t];
      printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
   }
   printf "\n";
}
```

Figure 13-2: Generating a formatted sales table with nested loops (`steelT.tab1`).

the choice is the same every time; the Figure 13-2 script relies on this consistency to ensure that all of the entries in one column of the table refer to the same product.

13.3 Iterating subject to a condition: the `repeat` statement

A second kind of looping construct, the `repeat` statement, continues iterating as long as some logical condition is satisfied.

Returning to the sensitivity analysis example, we wish to take advantage of a property of the dual value on the constraint `Time[3]`: the additional profit that can be realized from each extra hour added to `avail[3]` is at most `Time[3].dual`. When `avail[3]` is made sufficiently large, so that there is more third-week capacity than can be used, the associated dual value falls to zero and further increases in `avail[3]` have no effect on the optimal solution.

We can specify that looping should stop once the dual value falls to zero, by writing a `repeat` statement that has one of the following forms:

```
repeat while Time[3].dual > 0 {...};
repeat until Time[3].dual = 0 {...};
repeat {...} while Time[3].dual > 0;
repeat {...} until Time[3].dual = 0;
```

The loop body, here indicated by `{...}`, must be enclosed in braces. Passes through the loop continue as long as the `while` condition is true, or as long as the `until` condition is false. A condition that appears before the loop body is tested before every pass; if a `while` condition is false or an `until` condition is true before the first pass, then the loop body is never executed. A condition that appears after the loop body is tested after every pass, so that the loop body is executed at least once in this case. If there is no `while` or `until` condition, the loop repeats indefinitely and must be terminated by other means, like the `break` statement described below.

A complete script using `repeat` is shown in Figure 13-3. For this particular application we choose the `until` phrase that is placed after the loop body, as we do not want `Time[3].dual` to be tested until after a `solve` has been executed in the first pass. Two other features of this script are worth noting, as they are relevant to many scripts of this kind.

At the beginning of the script, we don't know how many passes the `repeat` statement will make through the loop. Thus we cannot determine `AVAIL3` in advance as we did in Figure 13-1. Instead, we declare it initially to be the empty set:

```
set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
```

and add each new value of `avail[3]` to it after solving:

```
model steelT.mod;
data steelT.dat;

option solution_precision 10;
option solver_msg 0;

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
param avail3_step := 5;

repeat {
   solve;
   let AVAIL3 := AVAIL3 union {avail[3]};
   let avail3_obj[avail[3]] := Total_Profit;
   let avail3_dual[avail[3]] := Time[3].dual;
   let avail[3] := avail[3] + avail3_step;
} until Time[3].dual = 0;

display avail3_obj, avail3_dual;
```

Figure 13-3: Script for recording sensitivity (`steelT.sa4`).

```
let AVAIL3 := AVAIL3 union {avail[3]};
let avail3_obj[avail[3]] := Total_Profit;
let avail3_dual[avail[3]] := Time[3].dual;
```

By adding a new member to `AVAIL3`, we also create new components of the parameters `avail3_obj` and `avail3_dual` that are indexed over `AVAIL3`, and so we can proceed to assign the appropriate values to these components. Any change to a set is propagated to all declarations that use the set, in the same way that any change to a parameter is propagated.

Because numbers in the computer are represented with a limited number of bits of precision, a solver may return values that differ very slightly from the solution that would be computed using exact arithmetic. Ordinarily you don't see this, because the `display` command rounds values to six significant digits by default. For example:

```
ampl: model steelT.mod; data steelT.dat; solve;
ampl: display Make;
Make [*,*] (tr)
:  bands  coils    :=
1   5990   1407
2   6000   1400
3   1400   3500
4   2000   4200
;
```

Compare what is shown when rounding is dropped, by setting `display_precision` to 0:

```
ampl: option display_precision 0;
ampl: display Make;
Make [*,*] (tr)
:         bands                coils           :=
1   5989.999999999999    1407.0000000000002
2   6000                 1399.9999999999998
3   1399.9999999999995   3500
4   1999.9999999999993   4200
;
```

These seemingly tiny differences can have undesirable effects whenever a script makes a comparison that uses values returned by the solver. The rounded table would lead you to believe that `Make["coils",2] >= 1400` is true, for example, whereas from the second table you can see that really it is false.

You can avoid this kind of surprise by writing arithmetic tests more carefully; instead of `until Time[3].dual = 0`, for instance, you might say `until Time[3].dual <= 0.0000001`. Alternatively, you can round all solution values that are returned by the solver, so that numbers that are supposed to be equal really do come out equal. The statement

```
option solution_precision 10;
```

toward the beginning of Figure 13-3 has this effect; it states that solution values are to be rounded to 10 significant digits. This and related rounding options are discussed and illustrated in Section 12.3.

Note finally that the script declares set `AVAIL3` as `default {}` rather than `= {}`. The former allows `AVAIL3` to be changed by `let` commands as the script proceeds, whereas the latter permanently defines `AVAIL3` to be the empty set.

13.4 Testing a condition: the `if-then-else` statement

In Section 7.3 we described the conditional (`if-then-else`) expression, which produces an arithmetic or set value that can be used in any expression context. The `if-then-else` *statement* uses the same syntactic form to conditionally control the execution of statements or groups of statements.

In the simplest case, the `if` statement evaluates a condition and takes a specified action if the condition is true:

```
if Make["coils",2] < 1500 then printf "under 1500\n";
```

The action may also be a series of commands grouped by braces as in the `for` and `repeat` commands:

```
if Make["coils",2] < 1500 then {
   printf "Fewer than 1500 coils in week 2.\n";
   let market["coils",2] := market["coils",2] * 1.1;
}
```

An optional `else` specifies an alternative action that also may be a single command:

```
    if Make["coils",2] < 1500 then {
       printf "Fewer than 1500 coils in week 2.\n";
       let market["coils",2] := market["coils",2] * 1.1;
    }
    else
       printf "At least 1500 coils in week 2.\n";
```

or a group of commands:

```
    if Make["coils",2] < 1500 then
       printf "under 1500\n";
    else {
       printf "at least 1500\n";
       let market["coils",2] := market["coils",2] * 0.9;
    }
```

AMPL executes these commands by first evaluating the logical expression following `if`. If the expression is true, the command or commands following `then` are executed. If the expression is false, the command or commands following `else`, if any, are executed.

The `if` command is most useful for regulating the flow of control in scripts. In Figure 13-2, we could suppress any occurrences of `100%` by placing the statement that prints `Sell[p,t]/market[p,t]` inside an `if`:

```
    if Sell[p,t] < market[p,t] then
       printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
    else
       printf "    --- ";
```

In the script of Figure 13-3, we can use an `if` command inside the `repeat` loop to test whether the dual value has changed since the previous pass through the loop, as shown in the script of Figure 13-4. This loop creates a table that has exactly one entry for each different dual value discovered.

The statement following `then` or `else` can itself be an `if` statement. In the formatting example (Figure 13-2), we could handle both `0%` and `100%` specially by writing

```
    if Sell[p,t] < market[p,t] then
       if Sell[p,t] = 0 then
          printf "         ";
       else
          printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
    else
       printf "    --- ";
```

or equivalently, but perhaps more clearly,

```
    if Sell[p,t] = 0 then
       printf "        ";
    else if Sell[p,t] < market[p,t] then
       printf "%7.1f%%", 100 * Sell[p,t]/market[p,t];
    else
       printf "    --- ";
```

```
model steelT.mod; data steelT.dat;
option solution_precision 10; option solver_msg 0;

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

let avail[3] := 1;
param avail3_step := 1;
param previous_dual default Infinity;

repeat while previous_dual > 0 {
   solve;
   if Time[3].dual < previous_dual then {
      let AVAIL3 := AVAIL3 union {avail[3]};
      let avail3_obj[avail[3]] := Total_Profit;
      let avail3_dual[avail[3]] := Time[3].dual;
      let previous_dual := Time[3].dual;
   }
   let avail[3] := avail[3] + avail3_step;
}

display avail3_obj, avail3_dual;
```

Figure 13-4: Testing conditions within a loop (`steelT.sa5`).

In all cases, an `else` is paired with the closest preceding available `if`.

13.5 Terminating a loop: `break` and `continue`

Two other statements work with looping statements to make some scripts easier to write. The `continue` statement terminates the current pass through a `for` or `repeat` loop; all further statements in the current pass are skipped, and execution continues with the test that controls the start of the next pass (if any). The `break` statement completely terminates a `for` or `repeat` loop, sending control immediately to the statement following the end of the loop.

As an example of both these commands, Figure 13-5 exhibits another way of writing the loop from Figure 13-4, so that a table entry is made only when there is a change in the dual value associated with `avail[3]`. After solving, we test to see if the new dual value is equal to the previous one:

```
if Time[3].dual = previous_dual then continue;
```

If it is, there is nothing to be done for this value of `avail[3]`, and the `continue` statement jumps to the end of the current pass; execution resumes with the next pass, starting at the beginning of the loop.

After adding an entry to the table, we test to see if the dual value has fallen to zero:

```
model steelT.mod;
data steelT.dat;

option solution_precision 10;
option solver_msg 0;

set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

let avail[3] := 0;
param previous_dual default Infinity;

repeat {
   let avail[3] := avail[3] + 1;
   solve;
   if Time[3].dual = previous_dual then continue;

   let AVAIL3 := AVAIL3 union {avail[3]};
   let avail3_obj[avail[3]] := Total_Profit;
   let avail3_dual[avail[3]] := Time[3].dual;

   if Time[3].dual = 0 then break;

   let previous_dual := Time[3].dual;
}

display avail3_obj, avail3_dual;
```

Figure 13-5: Using `break` and `continue` in a loop (`steelT.sa7`).

```
if Time[3].dual = 0 then break;
```

If it has, the loop is done and the `break` statement jumps out; execution passes to the `display` command that follows the loop in the script. Since the `repeat` statement in this example has no `while` or `until` condition, it relies on the `break` statement for termination.

When a `break` or `continue` lies within a nested loop, it applies only to the innermost loop. This convention generally has the desired effect. As an example, consider how we could expand Figure 13-5 to perform a separate sensitivity analysis on each `avail[t]`. The `repeat` loop would be nested in a `for {t in 1..T}` loop, but the `continue` and `break` statements would apply to the inner repeat as before.

There do exist situations in which the logic of a script requires breaking out of multiple loops. In the script of Figure 13-5, for instance, we can imagine that instead of stopping when `Time[3].dual` is zero,

```
if Time[3].dual = 0 then break;
```

we want to stop when `Time[t].dual` falls below 2700 for any `t`. It might seem that one way to implement this criterion is:

```
for {t in 1..T}
  if Time[t].dual < 2700 then break;
```

This statement does not have the desired effect, however, because `break` applies only to the inner `for` loop that contains it, rather than to the outer `repeat` loop as we desire. In such situations, we can give a name to a loop, and `break` or `continue` can specify by name the loop to which it should apply. Using this feature, the outer loop in our example could be named `sens_loop`:

```
repeat sens_loop {
```

and the test for termination inside it could refer to its name:

```
   for {t in 1..T}
      if Time[t].dual < 2700 then break sens_loop;
```

The loop name appears right after `repeat` or `for`, and after `break` or `continue`.

13.6 Stepping through a script

If you think that a script might not be doing what you want it to, you can tell AMPL to step through it one command at a time. This facility can be used to provide an elementary form of ‘‘symbolic debugger’’ for scripts.

To step through a script that does not execute any other scripts, reset the option `single_step` to 1 from its default value of 0. For example, here is how you might begin stepping through the script in Figure 13-5:

```
ampl: option single_step 1;
ampl: commands steelT.sa7;
steelT.sa7:2(18)   data ...
<2>ampl:
```

The expression `steelT.sa7:2(18)` gives the filename, line number and character number where AMPL has stopped in its processing of the script. It is followed by the beginning of the next command (`data`) to be executed. On the next line you are returned to the `ampl:` prompt. The `<2>` in front indicates the level of input nesting; ‘‘2’’ means that execution is within the scope of a `commands` statement that was in turn issued in the original input stream.

At this point you may use the `step` command to execute individual commands of the script. Type `step` by itself to execute one command,

```
<2>ampl: step
steelT.sa7:4(36)   option ...
<2>ampl: step
steelT.sa7:5(66)   option ...
<2>ampl: step
steelT.sa7:11(167)   let ...
<2>ampl:
```

If `step` is followed by a number, that number of commands will be executed. Every command is counted except those having to do with model declarations, such as `model` and `param` in this example.

Each `step` returns you to an AMPL prompt. You may continue stepping until the script ends, but at some point you will want to display some values to see if the script is working. This sequence captures one place where the dual value changes:

```
<2>ampl: display avail[3], Time[3].dual, previous_dual;
avail[3] = 22
Time[3].dual = 3620
previous_dual = 3620

<2>ampl: step
steelT.sa7:17(317)   continue ...
<2>ampl: step
steelT.sa7:15(237)   let ...
<2>ampl: step
steelT.sa7:16(270)   solve ...
<2>ampl: step
steelT.sa7:17(280)   if ...
<2>ampl: step
steelT.sa7:19(331)   let ...
<2>ampl: display avail[3], Time[3].dual, previous_dual;
avail[3] = 23
Time[3].dual = 3500
previous_dual = 3620
<2>ampl:
```

Any series of AMPL commands may be typed while single-stepping. After each command, the `<2>ampl` prompt returns to remind you that you are still in this mode and may use `step` to continue executing the script.

To help you step through lengthy compound commands (`for`, `repeat`, or `if`) AMPL provides several alternatives to `step`. The `next` command steps past a compound command rather than into it. If we had started at the beginning, each `next` would cause the next statement to be executed; in the case of the `repeat`, the entire command would be executed, stopping again only at the `display` command on line 28:

```
ampl: option single_step 1;
ampl: commands steelT.sa7;
steelT.sa7:2(18)   data ...
<2>ampl: next
steelT.sa7:4(36)   option ...
<2>ampl: next
steelT.sa7:5(66)   option ...
<2>ampl: next
steelT.sa7:11(167)   let ...
<2>ampl: next
steelT.sa7:14(225)   repeat ...
<2>ampl: next
steelT.sa7:28(539)   display ...
<2>ampl:
```

Type `next` *n* to step past *n* commands in this way.

The commands `skip` and `skip` *n* work like `step` and `step` *n*, except that they skip the next 1 or *n* commands in the script rather than executing them.

13.7 Manipulating character strings

The ability to work with arbitrary sets of character strings is one of the key advantages of AMPL scripting. We describe here the string concatenation operator and several functions for building up string-valued expressions that can be used anywhere that set members can appear in AMPL statements. Further details are provided in Section A.4.2, and Table A-4 summarizes all of the string functions.

We also show how string expressions may be used to specify character strings that serve purposes other than being set members. This feature allows an AMPL script to, for example, write a different file or set different option values in each pass through a loop, according to information derived from the contents of the loop indexing sets.

String functions and operators

The concatenation operator `&` takes two strings as operands, and returns a string consisting of the left operand followed by the right operand. For example, given the sets `NUTR` and `FOOD` defined by `diet.mod` and `diet2.dat` (Figures 2-1 and 2-3), you could use concatenation to define a set `NUTR_FOOD` whose members represent nutrient-food pairs:

```
ampl: model diet.mod;
ampl: data diet2.dat;
ampl: display NUTR, FOOD;
set NUTR := A B1 B2 C NA CAL;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR;
ampl: set NUTR_FOOD := setof {i in NUTR,j in FOOD} i & "_" & j;
ampl: display NUTR_FOOD;
set NUTR_FOOD :=
A_BEEF    B1_BEEF   B2_BEEF   C_BEEF    NA_BEEF   CAL_BEEF
A_CHK     B1_CHK    B2_CHK    C_CHK     NA_CHK    CAL_CHK
A_FISH    B1_FISH   B2_FISH   C_FISH    NA_FISH   CAL_FISH
A_HAM     B1_HAM    B2_HAM    C_HAM     NA_HAM    CAL_HAM
A_MCH     B1_MCH    B2_MCH    C_MCH     NA_MCH    CAL_MCH
A_MTL     B1_MTL    B2_MTL    C_MTL     NA_MTL    CAL_MTL
A_SPG     B1_SPG    B2_SPG    C_SPG     NA_SPG    CAL_SPG
A_TUR     B1_TUR    B2_TUR    C_TUR     NA_TUR    CAL_TUR;
```

This is not a set that you would normally want to define, but it might be useful if you have to read data in which strings like `"B2_BEEF"` appear.

Numbers that appear as arguments to `&` are automatically converted to strings. As an example, for a multi-week model you can create a set of generically-named periods `WEEK1`, `WEEK2`, and so forth, by declaring:

```
param T integer > 1;
set WEEKS ordered = setof {t in 1..T} "WEEK" & t;
```

Numeric operands to `&` are always converted to full precision (or equivalently, to `%.0g` format) as defined in Section A.16. The conversion thus produces the expected results for concatenation of numerical constants and of indices that run over sets of integers or constants, as in our examples. Full precision conversion of computed fractional values may sometimes yield surprising results, however. The following variation on the preceding example would seem to create a set of members `WEEK0.1`, `WEEK0.2`, and so forth:

```
param T integer > 1;
set WEEKS ordered = setof {t in 1..T} "WEEK" & 0.1*t;
```

But the actual set comes out differently:

```
ampl: let T := 4;
ampl: display WEEKS;
set WEEKS :=
WEEK0.1                   WEEK0.30000000000000004
WEEK0.2                   WEEK0.4;
```

Because 0.1 cannot be stored exactly in a binary representation, the value of `0.1*3` is slightly different from 0.3 in ‘‘full’’ precision. There is no easy way to predict this behavior, but it can be prevented by specifying an explicit conversion using `sprintf`. The `sprintf` function does format conversions in the same way as `printf` (Section A.16), except that the resulting formatted string is not sent to an output stream, but instead becomes the function’s return value. For our example, `"WEEK" & 0.1*t` could be replaced by `sprintf("WEEK%3.1f",0.1*t)`.

The `length` string function takes a string as argument and returns the number of characters in it. The `match` function takes two string arguments, and returns the first position where the second appears as a substring in the first, or zero if the second never appears as a substring in the first. For example:

```
ampl: display {j in FOOD} (length(j), match(j,"H"));
:    length(j) match(j, 'H')    :=
BEEF      4          0
CHK       3          2
FISH      4          4
HAM       3          1
MCH       3          3
MTL       3          0
SPG       3          0
TUR       3          0
;
```

The `substr` function takes a string and one or two integers as arguments. It returns a substring of the first argument that begins at the position given by the second argument; it has the length given by the third argument, or extends to the end of the string if no third argument is given. An empty string is returned if the second argument is greater than the length of the first argument, or if the third argument is less than 1.

As an example combining several of these functions, suppose that you want to use the model from `diet.mod` but to supply the nutrition amount data in a table like this:

```
param: NUTR_FOOD: amt_nutr :=
          A_BEEF      60
          B1_BEEF     10
          CAL_BEEF   295
          CAL_CHK    770
          ...
```

Then in addition to the declarations for the parameter `amt` used in the model,

```
set NUTR;
set FOOD;
param amt {NUTR,FOOD} >= 0;
```

you would declare a set and a parameter to hold the data from the ‘‘nonstandard’’ table:

```
set NUTR_FOOD;
param amt_nutr {NUTR_FOOD} >= 0;
```

To use the model, you need to write an assignment of some kind to get the data from set `NUTR_FOOD` and parameter `amt_nutr` into sets `NUTR` and `FOOD` and parameter `amt`. One solution is to extract the sets first, and then convert the parameters:

```
set NUTR = setof {ij in NUTR_FOOD}
                       substr(ij,1,match(ij,"_")-1);
set FOOD = setof {ij in NUTR_FOOD}
                       substr(ij,match(ij,"_")+1);
param amt {i in NUTR, j in FOOD} = amt_nutr[i & "_" & j];
```

As an alternative, you can extract the sets and parameters together with a script such as the following:

```
param iNUTR symbolic;
param jFOOD symbolic;
param upos > 0;
let NUTR := {};
let FOOD := {};

for {ij in NUTR_FOOD} {
   let upos := match(ij,"_");
   let iNUTR := substr(ij,1,upos-1);
   let jFOOD := substr(ij,upos+1);
   let NUTR := NUTR union {iNUTR};
   let FOOD := FOOD union {jFOOD};
   let amt[iNUTR,jFOOD] := amt_nutr[ij];
}
```

Under either alternative, errors such as a missing ‘‘_’’ in a member of `NUTR_FOOD` are eventually signaled by error messages.

AMPL provides two other functions, `sub` and `gsub`, that look for the second argument in the first, like `match`, but that then substitute a third argument for either the first occurrence (`sub`) or all occurrences (`gsub`) found. The second argument of all three of these functions is actually a *regular expression*; if it contains certain special characters, it is interpreted as a pattern that may match many sub-strings. The pattern `"^B[0-9]+_"`, for example, matches any sub-string consisting of a `B` followed by one or more digits and then an underscore, and occurring at the beginning of a string. Details of these features are given in Section A.4.2.

String expressions in AMPL commands

String-valued expressions may appear in place of literal strings in several contexts: in filenames that are part of commands, including `model`, `data`, and `commands`, and in filenames following > or >> to specify redirection of output; in values assigned to AMPL options by an `option` command; and in the string-list and the database row and column names specified in a `table` statement. In all such cases, the string expression must be identified by enclosing it in parentheses.

Here is an example involving filenames. This script uses a string expression to specify files for a `data` statement and for the redirection of output from a `display` statement:

```
model diet.mod;
set CASES = 1 .. 3;
for {j in CASES} {
   reset data;
   data ("diet" & j & ".dat");
   solve;
   display Buy >("diet" & j & ".out");
}
```

The result is to solve `diet.mod` with a series of different data files `diet1.dat`, `diet2.dat`, and `diet3.dat`, and to save the solution to files `diet1.out`, `diet2.out`, and `diet3.out`. The value of the index `j` is converted automatically from a number to a string as previously explained.

The following script uses a string expression to specify the value of the option `cplex_options`, which contains directions for the CPLEX solver:

```
model sched.mod;
data sched.dat;
option solver cplex;
set DIR1 = {"primal","dual"};
set DIR2 = {"primalopt","dualopt"};
for {i in DIR1, j in DIR2} {
   option cplex_options (i & " " & j);
   solve;
}
```

The loop in this script solves the same problem four times, each using a different pairing of the directives `primal` and `dual` with the directives `primalopt` and `dualopt`.

Examples of the use of string expressions in the `table` statement, to work with multiple database files, tables, or columns, are presented in Section 10.6.