11

Modeling Commands

AMPL provides a variety of commands like `model`, `solve`, and `display` that tell the AMPL modeling system what to do with models and data. Although these commands may use AMPL expressions, they are not part of the modeling language itself. They are intended to be used in an environment where you give a command, wait for the system to display a response, then decide what command to give next. Commands might be typed directly, or given implicitly by menus in a graphical user interface like the ones available on the AMPL web site. Commands also appear in scripts, the subject of Chapter 13.

This chapter begins by describing the general principles of the command environment. Section 11.2 then presents the commands that you are likely to use most for setting up and solving optimization problems.

After solving a problem and looking at the results, the next step is often to make a change and solve again. The remainder of this chapter explains the variety of changes that can be made without restarting the AMPL session from the beginning. Section 11.3 describes commands for re-reading data and modifying specific data values. Section 11.4 describes facilities for completely deleting or redefining model components, and for temporarily dropping constraints, fixing variables, or relaxing integrality of variables. (Convenient commands for examining model information can be found in Chapter 12, especially in Section 12.6.)

11.1 General principles of commands and options

To begin an interactive AMPL session, you must start the AMPL program, for example by typing the command **_ampl_** in response to a prompt or by selecting it from a menu or clicking on an icon. The startup procedure necessarily varies somewhat from one operating system to another; for details, you should refer to the system-specific instructions that come with your AMPL software.

Commands

If you are using a text-based interface, after starting AMPL, the first thing you should see is AMPL's prompt:

```
ampl:
```

Whenever you see this prompt, AMPL is ready to read and interpret what you type. As with most command interpreters, AMPL waits until you press the ''enter'' or ''return'' key, then processes everything you typed on the line.

An AMPL command ends with a semicolon. If you enter one or more complete commands on a line, AMPL processes them, prints any appropriate messages in response, and issues the `ampl:` prompt again. If you end a line in the middle of a command, you are prompted to continue it on the next line; you can tell that AMPL is prompting you to continue a command, because the prompt ends with a question mark rather than a colon:

```
ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];
```

You can type any number of characters on a line (up to whatever limit your operating system might impose), and can continue a command on any number of lines.

Several commands use filenames for reading or writing information. A filename can be any sequence of printing characters (except for semicolon `;` and quotes `"` or `'`) or any sequence of any characters enclosed in matching quotes. The rules for correct filenames are determined by the operating system, however, not by AMPL. For the examples in this book we have used filenames like `diet.mod` that are acceptable to almost any operating system.

To conclude an AMPL session, type `end` or `quit`.

If you are running AMPL from a graphical interface, the details of your interaction will be different, but the command interface is likely to be accessible, and in fact is being used behind the scenes as well, so it's well worth understanding how to use it effectively.

Options

The behavior of AMPL commands depends not only on what you type directly, but on a variety of options for choosing alternative solvers, controlling the display of results, and the like.

Each option has a name, and a value that may be a number or a character string. For example, the options `prompt1` and `prompt2` are strings that specify the prompts. The option `display_width` has a numeric value, which says how many characters wide the output produced by the `display` command may be.

The `option` command displays and sets option values. If `option` is followed by a list of option names, AMPL replies with the current values:

```
ampl: option prompt1, display_width;
option prompt1 'ampl: ';
option display_width 79;
ampl:
```

A `*` in an option name is a ''wild card'' that matches any sequence of characters:

```
ampl: option prom*;
option prompt1 'ampl: ';
option prompt2 'ampl? ';
ampl:
```

The command `option *`, or just `option` alone, lists all current options and values.

When `option` is followed by a name and a value, it resets the named option to the specified value. In the following example we change the prompt and the display width, and then verify that the latter has been changed:

```
ampl: option prompt1 "A> ", display_width 60;
A> option display_width;
option display_width 60;
A>
```

You can specify any string value by surrounding the string in matching quotes `'...'` or `"..."` as above; the quotes may be omitted if the string looks like a name or number. Two consecutive quotes (`''` or `""`) denote an empty string, which is a meaningful value for some options. At the other extreme, if you want to spread a long string over several lines, place the backslash character `\` at the end of each intermediate line.

When AMPL starts, it sets many options to initial, or default, values. The `prompt1` option is initialized to `'ampl: '`, for instance, so prompts appear in the standard way. The `display_width` option has a default value of 79. Other options, especially ones that pertain to particular solvers, are initially unset:

```
ampl: option cplex_options;
option cplex_options ''; #not defined
```

To return all options to their default values, use the command `reset options`.

AMPL maintains no master list of valid options, but rather accepts any new option that you define. Thus if you mis-type an option name, you will most likely define a new option by mistake, as the following example demonstrates:

```
ampl: option display_wdith 60;
ampl: option display_w*;
option display_wdith 60;
option display_width 79;
```

The `option` statement also doesn't check to see if you have assigned a meaningful value to an option. You will be informed of a value error only when an option is used by some subsequent command. In these respects, AMPL options are much like operating system or shell ''environment variables.'' In fact you can use the settings of environment variables to override AMPL's option defaults; see your system-specific documentation for details.

11.2 Setting up and solving models and data

To apply a solver to an instance of a model, the examples in this book use `model`, `data`, and `solve` commands:

```
ampl: model diet.mod; data diet.dat; solve;
MINOS 5.5: optimal solution found.
6 iterations, objective 88.2
```

The `model` command names a file that contains model declarations (Chapters 5 through 8), and the `data` command names a file that contains data values for model components (Chapter 9). The `solve` command causes a description of the optimization problem to be sent to a solver, and the results to be retrieved for examination. This section takes a closer look at the main AMPL features for setting up and solving models. Features for subsequently changing and re-solving models are covered in Section 11.4.

Entering models and data

AMPL maintains a ''current'' model, which is the one that will be sent to the solver if you type `solve`. At the beginning of an interactive session, the current model is empty. A `model` command reads declarations from a file and adds them to the current model; a `data` command reads data statements from a file to supply values for components already in the current model. Thus you may use several `model` or `data` commands to build up the description of an optimization problem, reading different parts of the model and data from different files.

You can also type parts of a model and its data directly at an AMPL prompt. Model declarations such as `param`, `var` and `subject to` act as commands that add components to the current model. The data statements of Chapter 9 also act as commands, which supply data values for already defined components such as sets and parameters. Because model and data statements look much alike, however, you need to tell AMPL which you will be typing. AMPL always starts out in ''model mode''; the statement `data` (without a filename) switches the interpreter to ''data mode'', and the statement `model` (without a filename) switches it back. Any command (like `option`, `solve` or `subject to`) that does not begin like a data statement also has the effect of switching data mode back to model mode.

If a model declares more than one objective function, AMPL by default passes all of them to the solver. Most solvers deal only with one objective function and usually select the first by default. The `objective` command lets you select a single objective function to pass to the solver; it consists of the keyword `objective` followed by a name from a `minimize` or `maximize` declaration:

```
objective Total_Number;
```

If a model has an indexed collection of objectives, you must supply a subscript to indicate which one is to be chosen:

```
objective Total_Cost["A&P"];
```

The uses of multiple objectives are illustrated by two examples in Section 8.3.

Solving a model

The `solve` command sets in motion a series of activities. First, it causes AMPL to generate a specific optimization problem from the model and data that you have supplied. If you have neglected to provide some needed data, an error message is printed; you will also get error messages if your data values violate any restrictions imposed by qualification phrases in `var` or `param` declarations or by `check` statements. AMPL waits to verify data restrictions until you type `solve`, because a restriction may depend in a complicated way on many different data values. Arithmetic errors like dividing by zero are also caught at this stage.

After the optimization problem is generated, AMPL enters a ''presolve'' phase that tries to make the problem easier for the solver. Sometimes presolve so greatly reduces the size of a problem that it become substantially easier to solve. Normally the work of presolve goes on behind the scenes, however, and you need not be concerned about it. In rare cases, presolve can substantially affect the optimal values of the variables — when there is more than one optimal solution — or can interfere with other preprocessing routines that are built into your solver software. Also presolve sometimes detects that no feasible solution is possible, and so does not bother sending your program to the solver. For example, if you drastically reduce the availability of one resource in `steel4.mod`, then AMPL produces an error message:

```
ampl: model steel4.mod;
ampl: data steel4.dat;
ampl: let avail['reheat'] := 10;
ampl: solve;
presolve: constraint Time['reheat'] cannot hold:
        body <= 10 cannot be >= 11.25; difference = -1.25
```

For these cases you should consult the detailed description of presolve in Section 14.1.

The generated optimization problem, as possibly modified by presolve, is finally sent by AMPL to the solver of your choice. Every version of AMPL is distributed with some default solver that will be used automatically if you give no other instructions; type `option solver` to see its name:

```
ampl: option solver;
option solver minos;
```

If you have more than one solver, you can switch among them by changing the `solver` option:

```
ampl: model steelT.mod; data steelT.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
```

```
ampl: reset;

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

ampl: option solver cplex;

ampl: solve;
CPLEX 8.0.0: optimal solution; objective 515033
16 dual simplex iterations (0 in phase I)

ampl: reset;

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

ampl: option solver snopt;

ampl: solve;
SNOPT 6.1-1: Optimal solution found.
15 iterations, objective 515033
```

In this example we reset the problem between solves, so that the solvers are invoked with the same initial conditions and their performance can be compared. Without reset, information about the solution found by one solver would be passed along to the next one, possibly giving the latter a substantial advantage. Passing information from one solve to the next is most useful when a series of similar LPs are to be sent to the same solver; we discuss this case in more detail in Section 14.2.

Almost any solver can handle a linear program, although those specifically designed for linear programming generally give the best performance. Other kinds of optimization problems, such as nonlinear (Chapter 18) and integer (Chapter 20), can be handled only by solvers designed for them. A message such as ''ignoring integrality'' or ''can't handle nonlinearities'' is an indication that you have not chosen a solver appropriate for your model.

If your optimization problems are not too difficult, you should be able to use AMPL without referring to instructions for a specific solver: set the solver option appropriately, type solve, and wait for the results.

If your solver takes a very long time to return with a solution, or returns to AMPL without any ''optimal solution'' message, then it's time to read further. Each solver is a sophisticated collection of algorithms and algorithmic strategies, from which many combinations of choices can be made. For most problems the solver makes good choices automatically, but you can also pass along your own choices through AMPL options. The details may vary with each solver, so for more information you must look to the solver-specific instructions that accompany your AMPL software.

If your problem takes a long time to optimize, you will want some evidence of the solver's progress to appear on your screen. Directives for this purpose are also described in the solver-specific instructions.

11.3 Modifying data

Many modeling projects involve solving a series of problem instances, each defined by somewhat different data. We describe here AMPL's facilities for resetting parameter values while leaving the model as is. They include commands for resetting data mode input and for resampling random parameters, as well as the `let` command for directly assigning new values.

Resetting

To delete the current data for several model components, without changing the current model itself, use the `reset data` command, as in:

```
reset data MINREQ, MAXREQ, amt, n_min, n_max;
```

You may then use `data` commands to read in new values for these sets and parameters. To delete all data, type `reset data`.

The `update data` command works similarly, but does not actually delete any data until new values are assigned. Thus if you type

```
update data MINREQ, MAXREQ, amt, n_min, n_max;
```

but you only read in new values for `MINREQ`, `amt` and `n_min`, the previous values for `MAXREQ` and `n_max` will remain. If instead you used `reset data`, `MAXREQ` and `n_max` would be without values, and you would get an error message when you next tried to solve.

Resampling

The `reset data` command also acts to resample the randomly computed parameters described in Section 7.6. Continuing with the variant of `steel4.mod` introduced in that section, if the definition of parameter `avail` is changed so that its value is given by a random function:

```
param avail_mean {STAGE} >= 0;
param avail_variance {STAGE} >= 0;

param avail {s in STAGE} =
   Normal(avail_mean[s], avail_variance[s]);
```

with corresponding data:

```
param avail_mean := reheat 35 roll 40 ;
param avail_variance := reheat 5 roll 2 ;
```

then AMPL will take new samples from the `Normal` distribution after each `reset data`. Different samples result in different solutions, and hence in different optimal objective values:

```
ampl: model steel4r.mod;
ampl: data steel4r.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
3 iterations, objective 187632.2489

ampl: display avail;
reheat  32.3504
  roll  43.038 ;

ampl: reset data avail;
ampl: solve;
MINOS 5.5: optimal solution found.
4 iterations, objective 158882.901

ampl: display avail;
reheat  32.0306
  roll  32.6855 ;
```

Only reset data has this effect; if you issue a reset command then AMPL's random number generator is reset, and the values of avail repeat from the beginning. (Section 7.6 explains how to reset the generator's ''seed'' to get a different sequence of random numbers.)

The let command

The let command also permits you to change particular data values while leaving the model the same, but it is more convenient for small or easy-to-describe changes than reset data or update data. You can use it, for example, to solve the diet model of Figure 5-1, trying out a series of upper bounds f_max["CHK"] on the purchases of food CHK:

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

ampl: solve;
MINOS 5.5: optimal solution found.
5 iterations, objective 74.27382022

ampl: let f_max["CHK"] := 11;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 73.43818182

ampl: let f_max["CHK"] := 12;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 73.43818182
```

Relaxing the bound to 11 reduces the cost somewhat, but further relaxation apparently has no benefit.

An indexing expression may be given after the keyword `let`, in which case a change is made for each member of the specified indexing set. You could use this feature to change all upper bounds to 8:

```
let {j in FOOD} f_max[j] := 8;
```

or to increase all upper bounds by 10 percent:

```
let {j in FOOD} f_max[j] := 1.1 * f_max[j];
```

In general this command consists of the keyword `let`, an indexing expression if needed, and an assignment. Any set or parameter whose declaration does not define it using an = phrase may be specified to the left of the assignment's `:=` operator, while to the right may appear any appropriate expression that can currently be evaluated.

Although AMPL does not impose any restrictions on what you can change using `let`, you should take care in changing any set or parameter that affects the indexing of other data.

For example, after solving the multiperiod production problem of Figures 4-4 and 4-5, it might be tempting to change the number of weeks `T` from 4 (as given in the original data) to 3:

```
ampl: let T := 3;
ampl: solve;
Error executing "solve" command:
error processing param avail:
        invalid subscript avail[4] discarded.
error processing param market:
        2 invalid subscripts discarded:
        market['bands',4]
        market['coils',4]
error processing param revenue:
        2 invalid subscripts discarded:
        revenue['bands',4]
        revenue['coils',4]
error processing var Sell['coils',1]:
        invalid subscript market['bands',4]
```

The problem here is that AMPL still has current data for 4th-week parameters such as `avail[4]`, which has become invalid with the change of `T` to 3. If you want to properly reduce the number of weeks in the linear program while using the same data, you must declare two parameters:

```
param Tdata integer > 0;
param T integer <= Tdata;
```

Use `1..Tdata` for indexing in the `param` declarations, while retaining `1..T` for the variables, objective and constraints; then you can use `let` to change `T` as you like.

You can also use the `let` command to change the current values of variables. This is sometimes a convenient feature for exploring an alternative solution. For example, here

ceil(x)	ceiling of x (next higher integer)
floor(x)	floor of x (next lower integer)
precision(x, n)	x rounded to n significant digits
round(x, n)	x rounded to n digits past the decimal point
round(x)	x rounded to the nearest integer
trunc(x, n)	x truncated to n digits past the decimal point
trunc(x)	x truncated to an integer (fractional part dropped)

Table 11-1: Rounding functions.

is what happens when we solve for the optimal diet as above, then round the optimal solution down to the next lowest integer number of packages:

```
ampl: model dietu.mod; data dietu.dat; solve;
MINOS 5.5: optimal solution found.
5 iterations, objective 74.27382022

ampl: let {j in FOOD} Buy[j] := floor(Buy[j]);
ampl: display Total_Cost, n_min, Diet_Min.slack;;
Total_Cost = 70.8

:       n_min Diet_Min.slack    :=
A         700          231
B1          0          580
B2          0          475
C         700          -40
CAL     16000         -640
;
```

Because we have used let to change the values of the variables, the objective and the slacks are automatically computed from the new, rounded values. The cost has dropped by about $3.50, but the solution is now short of the requirements for C by nearly 6 percent and for CAL by 4 percent.

AMPL provides a variety of rounding functions that can be used in this way. They are summarized in Table 11-1.

11.4 Modifying models

Several commands are provided to help you make limited changes to the current model, without modifying the model file or issuing a full reset. This section describes commands that completely remove or redefine model components, and that temporarily drop constraints, fix variables, or relax integrality restrictions on variables.

Removing or redefining model components

The `delete` command removes a previously declared model component, provided that no other components use it in their declarations. The form of the command is simply `delete` followed by a comma-separated list of names of model components:

```
ampl: model dietobj.mod;
ampl: data dietobj.dat;
ampl: delete Total_Number, Diet_Min;
```

Normally you cannot delete a set, parameter, or variable, because it is declared for use later in the model; but you can delete any objective or constraint. You can also specify a component ''name'' of the form `check` *n* to delete the *n*th `check` statement in the current model.

The `purge` command has the same form, but with the keyword `purge` in place of `delete`. It removes not only the listed components, but also all components that *depend* on them either directly (by referring to them) or indirectly (by referring to their dependents). Thus for example in `diet.mod` we have

```
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];
var Buy {j in FOOD} >= f_min[j], <= f_max[j];
minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];
```

The command `purge f_min` deletes parameter `f_min` and the components whose declarations refer to `f_min`, including parameter `f_max` and variable `Buy`. It also deletes objective `Total_Cost`, which depends indirectly on `f_min` through its reference to `Buy`.

If you're not sure which components depend on some given component, you can use the `xref` command to find out:

```
ampl: xref f_min;
# 4 entities depend on f_min:
f_max
Buy
Total_Cost
Diet
```

Like `delete` and `purge`, the `xref` command can be applied to any list of model components.

Once a component has been removed by `delete` or `purge`, any previously hidden meaning of the component's name becomes visible again. After a constraint named `prod` is deleted, for instance, AMPL again recognizes `prod` as an iterated multiplication operator (Table 7-1).

If there is no previously hidden meaning, the name of a component removed by `delete` or `purge` becomes again unused, and may subsequently be declared as the name of any new component of any type. If you only want to make some relatively limited modifications to a declaration, however, then you will probably find `redeclare` to be more convenient. You can change any component's declaration by writing the key-

word `redeclare` followed by the complete revised declaration that you would like to substitute. Looking again at `diet.mod`, for example,

    ampl: ***redeclare param f_min {FOOD} > 0 integer;***

changes only the validity conditions on `f_min`. The declarations of all components that depend on `f_min` are left unchanged, as are any values previously read for `f_min`.

A list of all component types to which `delete`, `purge`, `xref`, and `redeclare` may be applied is given in A.18.5.

Changing the model: `fix, unfix; drop, restore`

The simplest (but most drastic) way to change the model is by issuing the command `reset`, which expunges all of the current model and data. Following `reset`, you can issue new `model` and `data` commands to set up a different optimization problem; the effect is like typing `quit` and then restarting AMPL, except that options are not reset to their default values. If your operating system or your graphical environment for AMPL allows you to edit files while keeping AMPL active, `reset` is valuable for debugging and experimentation; you may make changes to the model or data files, type `reset`, then read in the modified files. (If you need to escape from AMPL to run a text editor, you can use the `shell` command described in Section A.21.1.)

The `drop` command instructs AMPL to ignore certain constraints or objectives of the current model. As an example, the constraints of Figure 5-1 initially include

    subject to Diet_Max {i in MAXREQ}:
        sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];

A `drop` command can specify a particular one of these constraints to ignore:

    drop Diet_Max["CAL"];

or it may specify all constraints or objectives indexed by some set:

    drop {i in MAXNOT} Diet_Max[i];

where `MAXNOT` has previously been defined as some subset of `MAXREQ`. The entire collection of constraints can be ignored by

    drop {i in MAXREQ} Diet_Max[i];

or more simply:

    drop Diet_Max;

In general, this command consists of the keyword `drop`, an optional indexing expression, and a constraint name that may be subscripted. Successive `drop` commands have a cumulative effect.

The `restore` command reverses the effect of `drop`. It has the same syntax, except for the keyword `restore`.

The `fix` command fixes specified variables at their current values, as if there were a constraint that the variables must equal these values; the `unfix` command reverses the effect. These commands have the same syntax as `drop` and `restore`, except that they name variables rather than constraints. For example, here we initialize all variables of our diet problem to their lower bounds, fix all variables representing foods that have more than 1200 mg of sodium per package, and optimize over the remaining variables:

```
ampl: let {j in FOOD} Buy[j] := f_min[j];
ampl: fix {j in FOOD: amt["NA",j] > 1200} Buy[j];
ampl: solve;
MINOS 5.5: optimal solution found.
7 iterations, objective 86.92
Objective = Total_Cost['A&P']

ampl: display {j in FOOD} (Buy[j].lb,Buy[j],amt["NA",j]);
:    Buy[j].lb    Buy[j]   amt['NA',j]   :=
BEEF      2         2          938
CHK       2         2         2180
FISH      2        10          945
HAM       2         2          278
MCH       2         9.42857   1182
MTL       2        10          896
SPG       2         2         1329
TUR       2         2         1397
;
```

Rather than setting and fixing the variables in separate statements, you can add an assignment phrase to the `fix` command:

```
ampl: fix {j in FOOD: amt["NA",j] > 1200} Buy[j] := f_min[j];
```

The `unfix` command works in the same way, to reverse the effect of fix and optionally also reset the value of a variable.

Relaxing integrality

Changing option `relax_integrality` from its default of 0 to any nonzero value:

```
option relax_integrality 1;
```

tells AMPL to ignore all restrictions of variables to integer values. Variables declared `integer` get whatever bounds you specified for them, while variables declared `binary` are given a lower bound of zero and an upper bound of one. To restore the integrality restrictions, set the `relax_integrality` option back to 0.

A variable's name followed by the suffix `.relax` indicates its current integrality relaxation status: 0 if integrality is enforced, nonzero otherwise. You can make use of this suffix to relax integrality on selected variables only. For example,

```
let Buy['CHK'].relax = 1
```

relaxes integrality only on the variable `Buy['CHK']`, while

```
let {j in FOOD: f_min[j] > allow_frac} Buy[j].relax := 1;
```

relaxes integrality on all `Buy` variables for foods that have a minimum purchase of at least some cutoff parameter `allow_frac`.

Some of the solvers that work with AMPL provide their own directives for relaxing integrality, but these do not necessarily have the same effect as AMPL's `relax_integrality` option or `.relax` suffix. The distinction is due to the effects of AMPL's problem simplification, or presolve, stage (Section 14.1). AMPL drops integrality restrictions *before* the presolve phase, so that the solver receives a true continuous relaxation of the original integer problem. If the relaxation is performed by the solver, however, then the integrality restrictions are still in effect during AMPL's presolve phase, and AMPL may perform some additional tightening and simplification as a result.

As a simple example, suppose that diet model variable declarations are written to allow the food limits `f_max` to be adjusted by setting an additional parameter, `scale`:

```
var Buy {j in FOOD} integer >= f_min[j], <= scale * f_max[j];
```

In our example of Figure 2-3, all of the `f_max` values are 10; suppose that also we set `scale` to 0.95. First, here are the results of solving the unrelaxed problem:

```
ampl: option relax_integrality;
option relax_integrality 0;

ampl: let scale := 0.95;
ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 122.89
6 MIP simplex iterations
0 branch-and-bound nodes
```

When no relaxation is specified in AMPL, presolve sees that all the variables have upper limits of 9.5, and since it knows that the variables must take integer values, it rounds these limits down to 9. Then these limits are sent to the solver, where they remain even if we specify a *solver* directive for integrality relaxation:

```
ampl: option cplex_options 'relax';
ampl: solve;
CPLEX 8.0.0: relax
Ignoring integrality of 8 variables.
CPLEX 8.0.0: optimal solution; objective 120.2421057
2 dual simplex iterations (0 in phase I)

ampl: display Buy;
Buy [*] :=
BEEF  8.39898
 CHK  2
FISH  2
 HAM  9
 MCH  9
 MTL  9
 SPG  8.93436
 TUR  2
;
```

If instead option `relax_integrality` is set to 1, presolve leaves the upper limits at 9.5 and sends those to the solver, with the result being a less constrained problem and hence a lower objective value:

```
ampl: option relax_integrality 1;
ampl: solve;
CPLEX 8.0.0: optimal solution; objective 119.1507545
3 dual simplex iterations (0 in phase I)

ampl: display Buy;
Buy [*] :=
BEEF  6.8798
 CHK  2
FISH  2
 HAM  9.5
 MCH  9.5
 MTL  9.5
 SPG  9.1202
 TUR  2
;
```

Variables that were at upper bound 9 in the previous solution are now at upper bound 9.5.

The same situation can arise in much less obvious circumstances, and can lead to unexpected results. In general, the optimal value of an integer program under AMPL's `relax_integrality` option may be lower (for minimization) or higher (for maximization) than the optimal value reported by the solver's relaxation directive.