phh(List) :- applist(write_space, List).
write_space(X) :- write(X), spaces(1).

Mapping is not restricted to lists, but can be defined for any kind of structure. For example, consider arithmetic expressions made up of functors such as * and +, each having two arguments. Suppose we wanted to map one expression into another, removing all multiplications by 1 in the process. One way to describe this algebraic simplification would be to define a predicate $s$ such that $s(0p, La, Ra, Ans)$ means that for an expression consisting of an operator $0p$ with a left argument $La$ and right argument $Ra$, a simplified form is the expression $Ans$. The facts for removing multiplications by 1 would look like this, with two facts accounting for the commutativity of multiplication:

\[
\begin{align*}
  s(\times, X, 1, X), \\
  s(\times, 1, X, X).
\end{align*}
\]

So, given an expression of the form $1 * X$, this table of simplifications could tell us to map it into whatever $X$ is. Let us see how we can use this in a program.

To simplify an expression $E$ using such a table of simplification rules, we need to first simplify the left-hand argument of $E$, then simplify the right-hand argument of $E$, and then see if the simplified result is in our table. If it is, we make the new expression whatever the table indicates. At the “leaves” of the expression tree there are integers or atoms, so we should use the built-in predicate atomic as a boundary condition to simplify leaves into themselves. As above, we can use “/..” to separate $E$ into its functor and components:

\[
\begin{align*}
  \text{simp}(E, E) &\ :- \text{atomic}(E), !. \\
  \text{simp}(E, F) &\ :- \\
                   &\ E =.. [0p, La, Ra], \\
                   &\ \text{simp}(La, X), \\
                   &\ \text{simp}(Ra, Y), \\
                   &\ s(0p, X, Y, F).
\end{align*}
\]

So, simp maps expression $E$ into expression $F$, using the facts found in a simplification table $s$. What happens if simp is presented with an operation for which no simplification can be made? To prevent $s(0p, X, Y, F)$ failing, we must have a “catchall” rule at the end of each operator’s part of the simplification table. The following simplification table includes rules for addition and multiplication, and shows the catchall rule for each operator included:

\[
\begin{align*}
  s(\times, X, 0, X). \\
  s(\times, 0, X, X). \\
  s(\times, X, Y, X+Y). /* catchall for + */ \\
  s(\times, X, Y, X). /* catchall for * */ \\
  s(\times, X, 0, 0).
\end{align*}
\]
\[
s(*, 0, _, 0).
\]
\[
s(*, 1, X, X).
\]
\[
s(*, X, 1, X).
\]
\[
s(*, X, Y, X*Y). /* catchall for */
\]

With the "catchall" rules present, there is now a choice of how to simplify some expressions. For instance, given 3+0, we can either use the first fact, or we can employ the "catchall" for +. Because of the way the facts are ordered, Prolog will always try the special case rules before the catchalls. Thus the first solution to simp will always be a true simplification (if there is one). However, alternative solutions will not be in the simplest possible form.

Another simplification used in computer-aided algebra is known as constant folding. The expression 3*4+a can have the constants 3 and 4 "folded" to form the expression 12+a. The folding rules can be added to the appropriate parts of the simplification table above. The rule for addition is

\[
s(+, X, Y, Z) :- \text{number}(X), \text{number}(Y), Z \text{ is } X+Y.
\]

The rules for the other arithmetic operations are similar.

In commutative operations such as multiplication and addition, the simplifications described above may have different effects on expressions that are written differently but are algebraically equivalent. For example, if a folding rule is available for multiplication, then the simp predicate will faithfully map \(2*3*a\) into \(6*a\), but \(a*2*3\) or \(2*a*3\) will be mapped into themselves. To see why this is, think about what the expressions look like as trees:

\[
2*3*a =
\]
\[
\begin{array}{c}
* \\
/ \\
* \\
/ \\
2 \\
3 \\
a
\end{array}
\]

\[
a*2*3 =
\]
\[
\begin{array}{c}
* \\
/ \\
* \\
/ \\
a \\
2 \\
3
\end{array}
\]

The first tree can have its bottom-most multiplication folded from \(2*3\) into \(6\), but the second tree has no sub-tree that can be folded. Because multiplication is commutative, adding the following rule to the table will suffice for this particular case:

\[
s(*, X*Y, W, X*Z) :- \text{number}(Y), \text{number}(W), Z \text{ is } Y*W.
\]

A more general algebra system can be constructed simply by adding more \(s\) clauses, instead of adding more programming to simp. Techniques for simplification, together