LA System User Manual

H.A. van der Meiden, W. Pasman, F.W. Jansen
Delft University of Technology

This project was made possible by the support of the SURF Foundation, the higher education and research partnership organisation for Information and Communications Technology (ICT). For more information about SURF, please visit www.surf.nl

Table of Contents

1. Introduction

The linear algebra training system (LA system) is a software tool for helping students improve their competence in linear algebra. It enables students to do exercises on the computer, e.g. solving a system of equations or proving a theorem, and gives intelligent feedback while the student is working on an exercise.

The student works on an exercise by applying rewrite rules to a given formula. A rewrite rule makes a calculation, based on the formula and input from the student, and replaces (part of) the formula with the result. For example, a rewrite rule may swap two rows in a matrix, or compute the Eigenvalues of a matrix. This way, the student can not make any calculation errors and saves a lot of time on tedious calculations. Thus, the student can now focus on deciding which rules to apply.

Furthermore, because the student can only use a predefined set of rewrite rules, the system can track the students actions, and give feedback during the exercise, i.e. whether the student is on the right track or hints on how to continue the exercise. To be able to give feedback, for each exercise a strategy should be defined, which describes the possible paths to the solution, i.e. sequences of rewrite rule applications, in a generic way.

2. Installation

The LA system requires Mathematica version 6 or later to run. Mathematica is available for various versions of Windows, Linux and OS X. For more information on Mathematica, see [www.wolfram.com].

The LA system software is a set of Mathematica packages (.m files) which should be installed in a directory that is in the Mathematica search path for packages. For more information about installing packages, see See Mathematica documentation.

Exercises are specified in Mathematica notebooks (.nb files). A set of exercises is provided with the LA system and should be installed in a single directory, and the name of this directory should be specified in the LA system configuration file.

The configuration file:

The LAsystem reads configuration setting from file `LAconfig.m.` Some configuration settings that can be edited are:

`ExercisesDirectory`: the directory where exercise notebooks can be found
`SolutionsDirectory`: the directory where student results (finished exercise notebooks) are saved.
`LogsDirectory`: the directory where log files (unfinished exercise notebooks) are saved.
`HintButton`: set StepButton = True to show Hint button in main menu.
`StepButton`: set StepButton = True to show Step button in main menu.
`SolutionButton`: set SolutionButton = True to show Solution button in main menu.
`DevelopmentButton`: set DevelopmentButton = True to show Dev button in main menu.
`AutoSelectAll`: if AutoSelectAll = True then, when user does not make a selection, select complete formula when apply is pressed.
`PunishHint:` if True, then if hint button pressed, exercise will not be marked as finished.
`PunishStep`: if True, then if step button pressed, exercise will not be marked as finished.
`PunishSolution`: If True, then if solution button pressed, exercise will not be marked as finished.

3. Basic use

Starting the LA system

To start the LA system:

- start Mathematica
- open the file `LAstart.nb` (using the menu File→Open)
- This notebook contains a button that starts the LA System.

Alternatively, first start Mathematica, and in a new notebook, type:

```
<< LAsystem`
```

and execute this command by pressing `Shift+Enter` (don't forget the back quote (`) at the end of the line).

Two new notebook windows will pop up: the `Working Notebook` and the `Button Notebook`. The working notebook shows a textual description of the exercise and a formula in which the user can make selections. The button notebook contains buttons that allow the student to manipulate the formula in the working notebook.

Doing exercises.

An exercise can be loaded by pressing the `Exercises` button. Whenever a user finishes an exercise, quites or loads a new exercise, the current working notebook is saved so it can be inspected later (by a teacher). Finished notebooks are saved in the `SolutionsDirectory` set in the configuration file. Unfinished notebooks are saved in the `LogsDirectory` set in the configuration file. Exercises that have been finished will be marked “Finished” in the list of exercises shown in the LA system.

A student works on an exercise by applying rewrite rules to a formula. By doing so, the formula is transformed into one that answers a problem stated in the description of the exercise. For example, if an exercise asks for a student to solve a system of equations, then the formula is a system of equations, e.g. $\{ x_1 + 3x_2 = 4, 2 x_1 - x_2 = 0\}$, and this formula should be transformed by the student into a set of trivial equations, i.e. of the form $\{x_1 = ..., x_2 = ...\}$.

To apply a rewrite rule, first a selection must be made in the working notebook. Then, in the button notebook, the `Apply` button should be pressed and a list of applicable rewrite rules will appear. Typically, a rewrite rule transforms the selected part of the formula, or manipulates a larger part of the formula using the selection as a parameter of the rewrite rules. The user may also be asked to enter parameters in a dialog that appears when a rewrite rule is selected.

Selections can be made with the mouse (or a similar input device). A selection is the entire formula or a part of the formula, e.g. a single equation or a row in a matrix. The formula is a nested expression,

containing numbers, variables, functions, matrices, definitions and other constructions. A selection is the entire formula or a part of the formula that is a proper sub-expression, i.e. it can not contain an unbalanced set of brackets.

When a rewrite rule has been applied, the description of the rule application is printed in the notebook, below the formula. A new formula, the result of the rewrite rule on the previous formula, is then printed below that. Selections can now be made in and rewrite rules can be applied to the new formula, and in general, only the bottom formula of the working notebook. Thus, a complete transcript of the students actions to solve a problem is created in the working notebook.

Consider the following example of a rewrite rule application. Figure 1 shows the working notebook with an exercise. The user has selected a set of equations, marked in black.

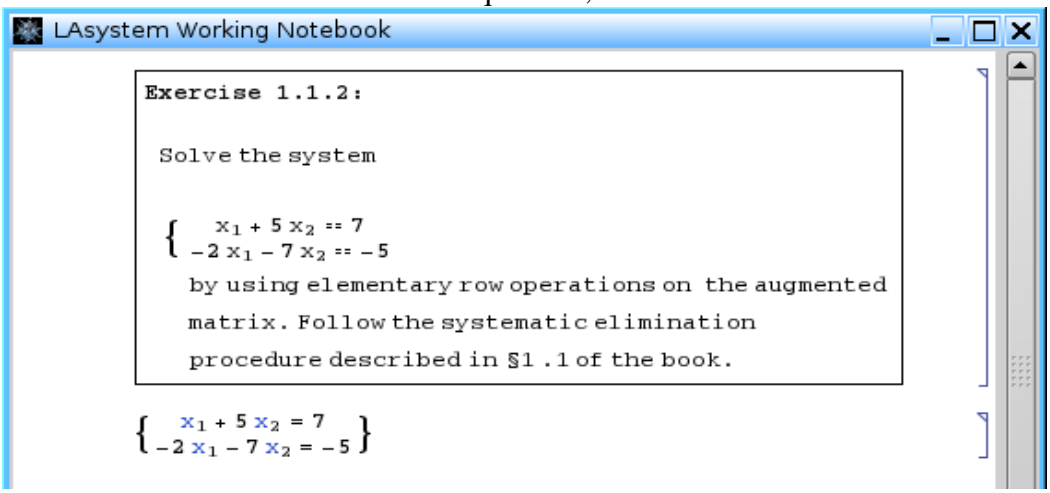

Figure 1: working notebook

In the button notebook, shown in Figure 2, after the `Apply rule` button is pressed, a number of applicable rewrite rules are shown.

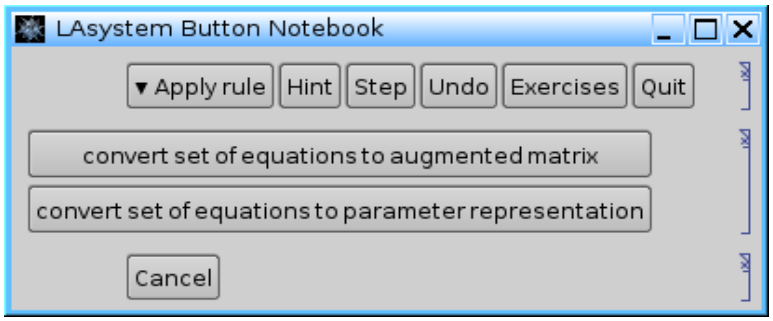

Figure 2: button notebook

Clicking on the rule: “convert set of equations to augmented matrix”, results in an updated working notebook, show in Figure 3.

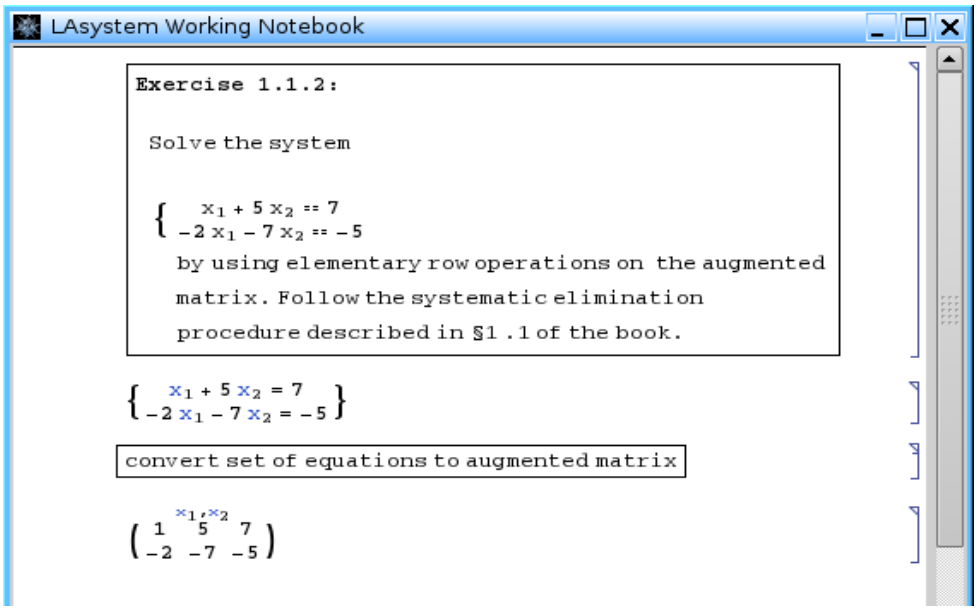

Figure 3: updated working notebook

To undo a rewrite-rule applications, use the `Undo` button.

Feedback

The LA system tracks the progress of the student during the exercise. The progress of the student is compared to a strategy that is defined with the exercise. The system determines when the student has finished the exercise, i.e. arrived at a correct answer. In that case, the system prints a message at the bottom of the working notebook stating that the exercise is finished, and no more selections can be made to apply rewrite rules. Note that there may be several answers, and several ways to arrive at an answer.

When the step taken by the student does not match the strategy, the system will inform the student that the step is not correct, and back-track by undoing the last performed step. It will however still show the result of the last step, so the user can (hopefully) see why the result is not correct.

When the student gets stuck, he or she may ask the system for a hint by pressing the `Hint` button. A dialog will be presented with a description of how to arrive at the answer. The hint system determines the hint from the exercise strategy and the previous rewrite rule applications. If more hints are requested, progressively more specific hints will be given.

The user can also ask the system to do the next step, by pressing the `Step` button. This may be repeated until the exercise is finished (and the student can try the next exercise on his/her own). Alternatively, pressing the `Solution` button will present a complete solution for the exercise, i.e. all steps are executed automatically.

4. System overview

To be able to create new exercises for the LA system, some knowledge of the design and implementation of LA system is needed. An overview of the system is provided here that will help the reader understand the following sections.

The LA system is a Mathematica application. It uses the Mathematica front-end to create a user interface and it uses Mathematica's pattern matching capabilities to implement rewrite operations on formulas in exercises.

The formulas manipulated by the rewrite rules are not expressed using the standard mathematical symbols and definitions supplied by Mathematica. In Mathematica, the expression Plus[3,4] automatically evaluates to 7. This is not desirable for formulas in the LA system, because formulas should be changed only by applying rewrite rules. Therefore, formulas are written using so-called *LA symbols*, e.g. instead of Plus[3,4], we write LAPlus[3,4].

LA symbols are defined in the file `LAterms.nb.` For most Linear Algebra exercises these symbols are sufficient, but for different domains, new symbols will have to be defined. Also, for displaying and reading these symbols in the Mathematica front end, definitions will need to be added to `LAconvert.nb.`

One part of the LAsystem that may need to be customized for new exercises is the set of rewrite rules available in the LAsystem. The current rule set contains rules for linear algebra, but other domains can also be added. Rewrite rules are defined in the file `basicrules.nb`.

Another important part of the LA system are the strategy definitions . A strategy language allows for a compact description of all possible solution paths for a type of exercise. This description is read and interpreted by the strategy parser, which is part of the system. When the user works on an exercise, the parser follows the steps taken and compares these steps with the strategy. The parser can decide whether the user is following the strategy, or not, and when the user deviates from the strategy, the system can provide a hint to guide the user in the right direction.

A solution path can be described as a sequence of rewrite steps. However, many different solutions paths may exist for a given exercise, e.g. some rewrite steps can be executed in any order. The strategy description should therefore be powerful enough to describe all solutions paths, in a compact way. The strategy language is based on a context-free grammar, extended with special control structures (the not and parallel operator). Strategies can also be re-used as sub-strategies for complex exercises.

All strategies available to the LA system are defined in the file `basicstrat.nb`. The current set of strategies is for linear algebra, but other domains can also be added.

The LA system source code consists of the following .nb files:

Site customisation:

LAconfig.nb	Configuration options
LAstart.nb	Simple start-up notebook

Definitions specific for Linear Algebra.

basicrules.nb	Rewrite rule definitions
basicstrat.nb	Strategy definitions
LAterms.nb	Formula symbol definitions
LAconvert.nb	Convert Mathematica expressions to LA system formula symbols

LA system source code:

LAsystem.nb	LAsystem main code, mostly user-interface stuff
LAtools.nb	Functions for doing Linear Algebra
ReadNotebook.nb	Functions for doing selection in workingnotebook
SimpleMatrixSelection.nb	Functions for selecting parts of matrices in formulas
Tools.nb	Misc functions
common.nb	Functions that are used everywhere, e.g. for error handling
hint.nb	Functions for generating hints
interfaces.nb	Functions for showing dialogs used by some rewrite rules
strategy.nb	Functions for managing database of strategies
wraprule.nb	Functions for managing database of rewrite rules

Strategy language and parser:

parstratparser.nb	Implements strategy language and parser
code.nb	Implements Code blocks for use in rewrite rules and strategies
unification.nb	Implements unification of attributed names and $variables

A lot of extra documentation can be found in the .nb files, and it is recommended to edit these files when modifying the LA system, instead of the .m files. After editing a .nb file, a new .m file is automatically created (Mathematica is set up to do this automatically by default). To see changes in the LAsystem, all definitions in the file need to be re-evaluated. The safest way to re-evaluate the definitions is by restarting the Mathematica kernel and re-leading the LAsystem package. However, in most cases it is sufficient to evaluate the modified notebook, or to re-load the modified package.

5. Adding and editing exercises

Exercises are specified in notebook files and can be edited using Mathematica. Exercises are defined by notebooks (.nb files) found in `ExecisesDirectory` (set in `LAconfig.m`). An exercises can be dynamic, i.e. have different numbers each time it is loaded, if the cells in the notebook use the 'code' style. The style of a cell can be set via the Mathematica front-end. A 'code' style cell will be displayed with a gray background.

An exercise notebook should contain two, three or four cells:

- The first cell contains a textual **description** of the exercise, which is copied to the working notebook when an exercise is loaded. If the cell uses the 'code' style, it will be evaluated and should yield a **box representation** of the description to be displayed.
- The second cell contains the **formula**, which is also copied to the working notebook. If the cell uses the 'code' style, it will be evaluated and should yield an **LA expression**
- The third cell is evaluated by Mathematica and should yield a list of **rewrite rule names**, which can be used by the student for this exercise. If this cell contains the symbol `All` or `AllWrapRules`, or is left empty, or does not exist, then all rewrite rules defined in the LA system will be available. This cell is evaluated by Mathematica so it can be used to define rewrite rules or to include files with rewrite rule definitions (see Section 6).
- The fourth cell is evaluated by Mathematica and should yield the start symbol of the **strategy** that is used for this exercise. This is generally of the form: `gterm[name, args...]`.If this cell is left empty (or does not exist) then no strategy is defined and no tracking and feedback will be available. This cell is also evaluated by Mathematica so it can be used to define grammars for strategies or to include files (see Section 7).

An example exercise notebook is shown in Figure 4. This example does not use the 'code' style.

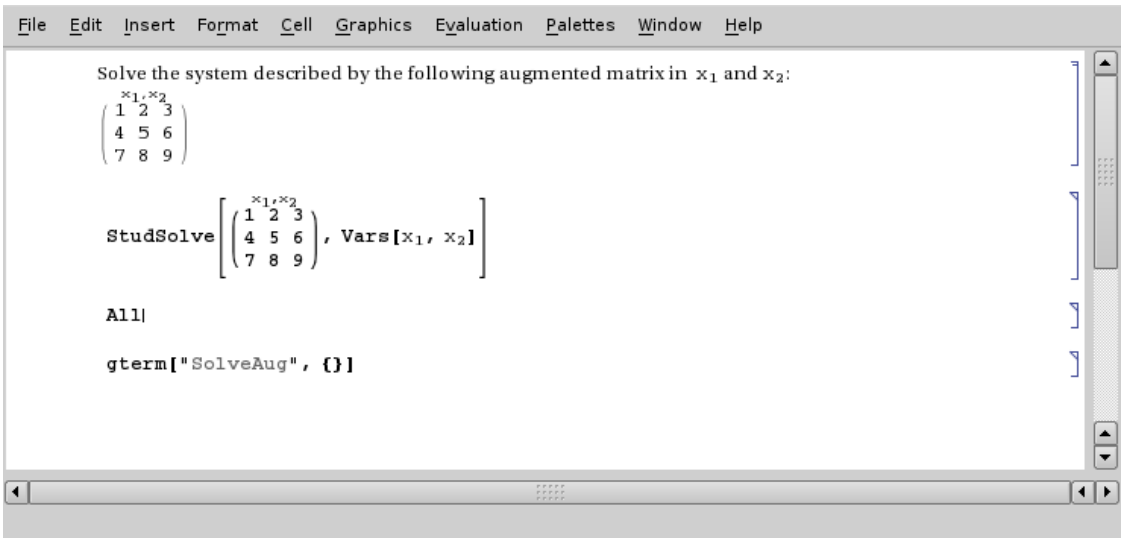

Figure 4: exercise notebook

Editing formulas graphically

Formulas in an exercise can be edited graphically (WYSIWYG) using the Mathematica front-end. In particular, for changing numbers in the formula around, this is easy.

For some mathematical constructs the notation used by the LA system is different from the notation used by Mathematica. In particular, a matrix in Mathematica can be represented simply by a block of numbers, but in the LA system, high round brackets are required around a matrix for it to be correctly recognized. A matrix with the names of variables above it is called an augmented matrix, and this notation is also frequently used in the LA system (e.g. in Figure 4).

Editing variables in a formula by hand is possible but tricky. In Mathematica, any symbol that is not defined can be used as a free variable. However, in the LA system, a real valued variable must be represented by `RealVar[name]`, where name is any string. To create variables with a subscript is a bit tricky; we need to use special string layout control characters: a variable x_1 is represented as: RealVar["\!\(x_1\)"]. Or, alternatively, we can use RealVar[Subscript[“x”, 1]]. In the exercise notebook, however, we can not see the RealVar notation, because it is encapsulated in a so called InterpretationBox, which allows Mathematica to maintain a separate representation and interpretation. To see the complete contents of the cell without formatting, press Ctrl+Shift+E (and again to switch back to normal view). Now, both the representation and interpretation of an InterpretationBox can be edited.

To make entering and editing formulas easier, two palettes are bundelled with the LA system, called `LA system variables` and `LA system matrices,` which provide buttons for entering variables with subscripts, special symbols and matrices. These palettes can be installed from Mathematica, or by

placing them in a special directory (see Mathematica documentation on Installing Palettes).

LA symbols and LA expressions

Instead of graphically entering or editing formulas in the exercise notebook, as described above, it is also possible to enter a formula using the formal notation used internally in the LA system, i.e. using LA symbols, to construct an LA expression. This also allow us to create exercises with generated content, such as exercises of the same type but with different numbers.

For example, the formula in Figure 4, is represented internally by the following LA expression:

```
AugmentedMatrix[
     LAMatrix[{1, 2, 3}, {4, 5, 6}, {7,8,9}],
     {RealVar[Subscript[“x”,1]], RealVar[Subscript[“x”,1]]}
]
```

This expression can be entered in the second cell of a Mathematica notebook, if this cell uses the 'code' style. These cells are displayed with a gray background. (Press Ctrl+8 to change a cell to the 'code' style).

Most mathematical symbols defined in Mathematica, cannot be used to represent formulas in the LA system, because Mathematica will evaluate those expressions. For example, the symbol `Plus` is already defined in Mathematica, and `Plus[1,1]` would instantly be rewritten to 2. Instead, use the term `LAPlus[1,1],` which will appear in the LA system as 1+1.

The terms defined by the LA system are described in Appendix A. New terms can also be defined, in the file `LAterms.m,` and conversion rules for displaying LA expressions re defined in `LAconvert.m`.

To convert between regular Mathematica expressions and LA expressions, use the `Exp2L` and `L2Exp` functions, e.g.:

```
Exp2L[Plus[1,1]] → LAPlus[1,1]
L2Exp[LAPlus[1,1]] → Plus[1,1] → 2
```

For creating variables and parameters with subscripts, use the `Subscript symbol`:

`RealVar[Subscript[“x”,2]]  will display as` x_2

6. Adding and editing rewrite rules

The LA system defines a standard set of rewrite rules in the file `basicrules.m`. This file is loaded when the LA system is loaded. An overview of the basic rules implemented in the LA system is given in Appendix B.

New rules may also be defined in a exercise notebook, in the 3rd cell, which is also used to specify the rule set that should be available for the specific exercise. This cell is evaluated by Mathematica, and can therefore contain a complete program that defines new rules and eventually returns a list of rule names.

Rewrite rules

A rewrite rule consists of two parts:

1. An attributed name (gterm), used to refer to the rule
2. A rule with a position (RuleAtPos), consisting of a **pattern**, a **position** and a **result**.

In Mathematica, a rewrite rule is represented as:

```
RewriteRule[
      gterm["name", attributes...],
      RuleAtPos[pattern, position, result]
]
```

A pretty notation for the rule is:

```
name(attributes...): pattern @position → result
```

This pretty notation can be generated by the LA system, but currently can not be used to define rules.

Attributed names

An attributed name is a structure of the form: name[attributes...], which is used to refer to a rewrite rule or a grammar rule (see Section 7). An attributed name is represented internally (in Mathematica code) by a **gterm** structure: `gterm[name, attributes...]`.

The name of a rewrite rule must be a lower case string (as opposed to names of grammar rules, which must start with a capital letter). Attributes of a rule may be $-variables, i.e. symbols starting with a $ and that do not contain any upper case characters, e.g. `$row` and `$x` are valid attribute names, but `$Row` and `x` are not (these symbols are treated as constants). A rewrite rule can be instantiated by referring to the attributed name, where $-variables are replaced by values. Rewrite rules may be completely instantiated (e.g. `somerulename[2,2]`), may be left uninstantiated (e.g. `somerulename[$x,$y]`) or instantiated partially (e.g. `somerulename[1,$z]`).

The same $-variable can be used more than once in the attributed name of a rewrite rule, e.g. we can define `somerulename[$x,$x]`. Attributes are unified with the 'function call'. For instance, if `somerulename[1,$y]` is called, `$x` would be set to 1, and if the rule can be applied, on return `$y` would also be set to 1.

Pattern

A rewrite rule is applicable to any expression that matches the specified pattern. The pattern is a Mathematica expression with blanks and pattern variables, and is matched with the expression by Mathematica.

Basically, a pattern is an expression that contains so called blanks and pattern variables. A blank is represented by an underscore (_) and can match any expression. A pattern variable is a symbol ending with an underscore (e.g. x_) and when matching the variable is assigned a value corresponding to the matched expression. The pattern variable can be reused (without the underscore) in the result of the rewrite rule. A blank followed by a symbol (e.g. _LAMatrix) matches only expressions with a given head. A sequence of two blanks (__) matches any sequence of 1 or more expressions and a sequence of three blanks (___) matches any sequence of zero or more expressions (see Mathematica documentation for more details on matching).

Consider the following rewrite rule:

```
myrule:x_LAMatrix → Transpose[x]
```

This rule matches only with expressions of the form LAMatrix[...] and assigns the matched expression to the variable x. When the rule is applied, it replaces the matched expression with the result of Transpose[x].

Position

The position argument of a rule specifies to which subterm of the current term (in the formula/expression workingnotebook) the rule should be applied. Positions are specified using the conventions of Mathematica (see Mathematica documentation on Position). The pos argument can be a $-variable. If no value is associated with the pos argument, then the LA system will search for a matching subexpression in the current term. If a position is given (by the user or the strategy parser) then the rule is applied only if the specified subterm matches the pattern. If an explicit position (instead of a $-attribute) is given in the definition of the rule, then the rule is always applied at the same position in the current term, if that subterm matches the pattern.

Result

The result of a rule is a term that replaces the term on which the rule was applied (a subterm of the

current term, specified by the position argument). The result may be any Mathematica expression (although, in the LAsystem, it is typically an LA expression) and it may refer to the variables specified in the pattern. The expression will be evaluated only when the rule matches.

Also, the result may be **Code block**. A code block may contain arbitrary Mathematica code, which will be evaluated when the rewrite rule is applied, and only inside the code block will $-variables be evaluated, i.e. the $-variables specified in the gterm and the $-variables that specifies the position. For example, `Code[$x]` returns the value of `$x(`if a value is defined for `$x`, otherwise the symbol $x is returned).

When the result of a rewrite rule is the special value **$Failure**, the rewrite rule is not applied, i.e. the term is not replaced. Also, if an error occurs in a Code block, $Failure is returned and thus the rule is not applied. It is recommended to always use a Code block to specify the result of a rule (except perhaps for very simple rules) because it guarantees that Mathematica functions are not evaluated before rule application and any errors that may occur during rule application are correctly handled.

Here is an example of a rewrite rule implemented in the LA system:

```
RewriteRule[gterm["swaprows", $r1, $r2],
      RuleAtPos[
            AugmentedMatrix[mat_LAMatrix, vars_],
            $pos,
            Code[AugmentedMatrix[SwapRows[mat, $r1, $r2], vars]]
      ]
]
```

This rule, named “swaprows” swaps two rows in an augmented matrix. Its attributes are the row numbers of the rows to be swapped. The pattern specifies that the rule is applicable to a term that is an `AugmentedMatrix` containing an `LAMatrix` (assigned to a variable mat), and a list of variables (assigned to variable vars). The position in the current term where the rule is applied is automatically determined, as `$pos` is not an attribute of the gterm. The result is a Code block, in which `$r1` and `$r2` will be replaced by values when the rule is applied, and these values are passed to the function `SwapRows` (defined by the LA system). A new `AugmentedMatrix` is constructed with a new `LAMatrix`, the result of `SwapRows`, and the same variables as the original term.

User interface definition

For the user to be able to apply rules, a user interface must be created for each rule. This is done with the **WrapRule** function. Calling `WrapRule[rule, description]` creates a basic user-interface for a rule: When a selection is made in the working notebook that matches the pattern defined in the rule, then the rule becomes available in the button notebook (after the user has pressed the apply button).

Many rules require additional parameters to be specified by the user. When a rule is applied, it calls the function `AskParameters[rulename, currentterm]`, where rulename is the name of the applied rule

and currentterm is the term in the working notebook, including the selection. The selection is a subterm of the current term, represented as `Focus[selection]` where, selection is the subterm that was selected. For example. if the current term is: `LAPlus[Focus[LATimes[3,4]],5]`, then the complete formula is 3*4+5 and the selection is 3*4.

The `AskParameters` function can be redefined for each rule, and can determine parameter values by asking the user for input (via dialogs) or determine parameter values from the current term (the context of the rule application). The `AskParameters` function should return a list of parameter values, for subsequent attributes specified in the gterm of a rewrite rule. If it returns anything other than a list, the rule application is canceled.

For example, for the "swaprows" rule, the following `AskParameters` function is defined as:

```
AskParameters["swaprows", mat_LAMatrix] := Module[{oldrow, newrow, matlen},
   oldrow = Position[mat, Focus][[1, 1]];
   newrow = AskRowNumber[
     "swap row number " <> ToString[oldrow] <> " with row number:",
     Length[mat]];
   If[newrow === Null || oldrow == newrow, Null, {oldrow, newrow}]
  ];
```

By default, a rule is applicable only if the selection matches the pattern specified in the rewrite rule. It is possible to activate rules with other selections, for example, a swap rows operation in a matrix can be applied when selecting a row in the matrix (note: the rule rewrites the whole matrix). For such rules, we can redefine the `WrapPattern[rulename]` function. This function should return a pattern containing an `Focus` term. If a match is found, then the rule is applied to the matching subterm of the current term.

The "swaprows" rule can be applied when a row in a matrix is selected instead of the entire matrix. Therefore, the `WrapPattern` for this rule is defined as :

```
WrapPattern["swaprows"] := LAMatrix[___, Focus[_], ___];
```

Automatic rules

These rules are special in that it is automatically executed when specified by a strategy, or, if no strategy is specified, then it is automatically executed whenever a term can be simplified. Thus, the user of the LA system never explicitly applies these rules. If the rule is properly defined, it should not be applicable when it has already been applied once, and thus should not visible in the list of applicable rules. The LA system will execute automatic rules only once, even if applicable several times, thus preventing infinite repetition.

Automatic rules can be defined by evaluating

```
IsAutomaticRule[name_]=True
```

7. Adding and editing strategies

A strategy is a set of permissible rewrite sequences on a given term. A strategy is described in a strategy language, consisting of grammar rules, which describe how to generate such sequences. The strategy language is based on a context free grammar (CFG), where the terminals of the CFG refer to the available rewrite rules, and a "string" produced by the CFG is a rewrite sequence. This basic CFG was extended with attributes, with a not operator and with a parallel operator, to fill in the need for more powerful notations to enable compact notation of strategies.

To set the strategy that should be followed for a given exercise, a single gterm is specified in the 4th cell of the corresponding exercise notebook. This gterm is the start-symbol of the strategy, typically its name describes the strategy and its attributes specify details, such as the sub-term on which the strategy is to be applied, or a variant of the strategy, e.g. `SolveLinearEqs[$pos, $vars]`. The start-symbol should be a gterm that appears (or can be unified by variable substitution with a gterm that appears) on the left-hand side of a grammar rule.

The grammar rules of the strategy can be specified in the file `basicstart.m` or in the 4th cell of the exercise notebook (note that the cell should still evaluate to a gterm, to be used as the start-symbol of the strategy for that exercise).

Grammar rules

A grammar rule defines a grammar term, specified by an attributed name (gterm) on the left hand side, and a sequence of terms on the right hand side. Each term on the right hand side (rhs-term) can be one of:

- An attributed name (`gterm[Name,...]`) that matches a **grammar rule or rewrite rule** definition;
- A **Code block**: `Code[...]`;
- A **not operator:** `not[gterm[...]]` where again the gterm matches with some rule;
- A **parallel operator**: `par[gterm[...], gterm[...]]` where both gterms match with some rule;
- *(experimental!)* A **finish** operator: `finish[gterm[...]]` where gterm matches a grammar rule

The attributed name (gterm) at the left-hand side of a grammar rule must be a string that starts with a capital letter, e.g. “SolveLinearEqs”. On the right hand-side, attributed names (gterms) may appear that refer to other grammar rules (names starting with the first letter capitalized) or rewrite rules (lowercase names).

The notation for a grammer rule used in this document is as follows:

```
Name(args...) := rhs-terms...
```

The pretty notation can be generated by the strategy module, but cannot (currently) be used to input grammar rules; instead, a grammar rule is represented in Mathematica as:

```
GrammerRule[ gterm[“Name”, …], { rhs-terms… }]
```

where rhs-term is one of: `gterm[...]`, `not[gterm[...]]` or `par[gterm[...], gterm[...]]`.

To add a grammar rule to the rule set that is kept by the LA system, use the following Mathematica expression (which takes the same arguments as a GrammarRule definition):

```
AddGrammerRule[ gterm[“Name”, …], { rhs-terms… }]
```

The meaning of a grammar rule

In short, a grammar rule specifies a strategy that corresponds to a sequence of sub-strategies and/or rewrite rule applications.

More precisely, a grammar rule specifies that the attributed name on the left-hand-side (a strategy), is applicable if all the terms in the sequence on the right-hand-side are applicable, in the given order.

Note that 'applicable' has a different meaning for grammar rules (strategies) then for rewrite rules: a grammar rule is applicable to a sequence of grammar terms, and rewrite rule is applicable to a formula term. A grammar term that refers to a rewrite rule is applicable when the rewrite rule has been applied by the user. (It would perhaps be better to say that a grammar rule is recognized by a sequence of terms, instead of applicable to a sequence of terms). To apply a strategy means to apply some sequence of rewrite rules such that the strategy is applicable. There may be many such sequences, or none.

When a rewrite rule is applied by the user, the attributed name of the rule (with any $variables replaced by values set by the user) is sent to the strategy parser that is part of the LA system. The parser then determines which strategies are applicable to the current sequence of rewrite rule applications, i.e. if all the terms on the right hand side of a grammar rule are applicable, then the grammer term on the right hand side is applicable.

The applicable grammar terms are stored in a parse state, which is incrementally updated every time a rewrite rule is applied. If the start-symbol (the strategy specified for the exercise) is applicable, then the parse state is **finished**. The parser can also determine whether it is still possible to finish the parse, by trying all possible rewrite rule applications specified by the strategy. If so, then the parse is **on-track**, otherwise the parse is **dead**.

To give an example, consider the following strategy:

```
S:=r1 r2
```

Strategy `S` is applicable if the sequence `r1 r2` is applicable. The terms r1 and r2 refer to rewrite rules, and therefore, for `S` to be applicable, the user must first apply rule `r1` and then apply rule `r2`.

Several grammar rules with the same left-hand-side are interpreted as alternative strategies.
For example:

```
S:=r1
S:=r2
```

This means that the strategy `S` is applicable if `r1` is applicable or `r2` is applicable. Thus, for S to be applicable, the user must apply rule `r1` or rule `r2`, but not both, because then sequence the sequence `r1 r2` is applicable, not the sequence `r1` or the sequence `r2`).

A **Code** block in a grammar rule is applicable if it does not return $Failure. A Code block is immediately executed by the parser as soon as it needs it to recognize a strategy. If the Code block returns $Failure, the associated parse attempt fails and the corresponding strategy is not applicable. But all other return values are *ignored.* A Code block can be used for its side effects: setting and clearing attribute values, or for letting a strategy fail.

The **not[X]** operator checks that a attributed name X can *not* be applied, i.e. not[X] can be applied if X can not be applied. The not operator is slightly special: whenever the parser is testing a not, rewrite rules can be called without having the $pos variable being set. If it is not set, the parser will try all potential positions where the rule can be applied. For each applicable position, $pos will be set accordingly[1] and further parsing is tried.

The **par[X,Y]** operator checks that attributed name X and Y can be applied in parallel (in other words, interleaved). Either X or Y can "eat" the the next rewrite actions of the user. It is possible to share attributes between X and Y. Shared attributes are those attributes that occur in both X and Y within the par[]. So in the call `par(T($x,5,aap($w),$z), U($z,ga($w),$c) )` the attributes $z and $w occur in both heads and thus are shared between T and U. This way, the parsers for X and Y can "communicate" about their progress and influence the other parser.

Note: The parser internally needs to keep track of all possible in-between results of X and Y. With excessive ambiguity, the parser may run out of memory. Therefore it is important to minimize ambiguity of X and Y. This problem can also occur if one parser makes multiple assignments to a variable. As with ambiguity, all these possible assignments have to be tracked separately and excessive

[1] This is not possible during normal parsing, as each different application position requires a separate parser.

occurrences may cause excessive memory and CPU usage.

(experimental!) The **finish[X]** operator shortcuts a strategy by asserting that is is applicable. It is assumed that finish[X] occurs only inside strategy of X. When finish[X] is encountered by the parser, any remaining, not yet applied sub-strategies of X are skipped, and X is marked as if it had been completely applied. Thus any strategy depending on X can be continued. This operator is useful to 'split' a strategy during its execution.

Note: the finish operator is a bit of a hack, and has not been fully tested. It may cause problems when encountered during a parallel parsing situation. It does not fit well in concept of a context-free grammar; rather it is something that is possible because the strategy parser is a so-called chart parser (see strategy parser manuals).

Attributes and $-variables

A term referring to a rewrite rule is applicable if the rewrite rule has been succesfully applied, and if all $-variable attributes have a value, However, terms referring to grammar rules (strategies) can be applicable even with uninstantiated $-variable attributes. A grammar term with uninstantiated $-variables is applicable if there is an applicable grammar term that can be **unified** with it, i.e. if there is an assignment for the variables such that the terms is equal to an applicable term.

For example, consider the following strategy:

```
S:=T[$x]
T[$y]:=Code[$y=1]
```

Strategy S is applicable if T[$x] is applicable. T[$y] is applicable if the Code block is applicable. The Code block is in fact applicable, because it does not return $Fail, Also, the code block sets $y=1 and the parser determines that therefore T[1] is applicable. T[$x] can be unified with T[1] by unification, i.e. by assigning $x=1, and therefore S is applicable.

A special variable **$CurrentTerm** can be used in Code blocks that refers to the formula term at the appropriate point in the rewriting sequence as recognized by the stategy. So, for example, given the following strategy:

```
S:=Code[Print[$CurrentTerm]], add1, Code[Print[$CurrentTerm]]
```

Starting a parse with a formula term “100”, the parser first prints “100” and then waits for the user to apply the rule `add1`. After the user has applied the correct rule, the parser prints “101”.

Writing strategies

With only basic operators (par and not) it is somewhat difficult to write complex strategies; strategies can become long and difficult to read. Strategy descriptions are not always intuitive because they specify not 'how' to apply a strategy, but 'when' a strategy can be applied. Also, there are difficulties due to the syntax of the strategy language. This results in strategies containing many sub-strategies that do very little actual 'computation', but are needed for appropriate handling of conditions on attribute values.
Therefore, some commonly used control-structures from other programming languages have been implemented as strategies that can be used to write more easily readable strategies (see also Appendix C).

RepExh

RepExh[$s] repeatedly applies the strategy $s, until it fails.

For example, consider the following strategy:

```
sub1: x -> x-1
DoSub1: {Code[If[x>0, $Failure]], sub1}
T:=RepExh[DoSub1]
```

With `$CurrentTerm = 3`, `T` can be applied if the following sequence of rewrite rule has been applied`: {sub1,sub1,sub1}`.

The implementation of RepExh is simple and instructive:

```
RepExh[$s] := $s RepExh[$s]
RepExh[$s] := not[$s]
```

The first line states that RepExh[$s] can be applied if $s can be applied, and subsequently RepExh[$s] can be applied recursively. The second line states that when $s fails, not[$s] must be applicable, and therefore RepExh[$s] can be successfully applied. Effectively, RepExh[$s] repeatedly applies $s until application of $s fails (RepExh[$s] does not fail, even if $s cannot be applied even once).

If

If[$condition, $s1, $s2] applies strategy $s1 if condition evaluates to True or else applies strategy $s2. Note: $condition must be a Code block (or any other holded expression, e.g. Hold or HoldComplete).

For example: consider the following stategy:

```
T:=If[Code[$CurrentTerm>1], GoDown, GoUp]
```

Suppose that $CurrentTerm == 2, then to apply T, the user must apply strategy GoDown. Suppose that $CurrentTerm == 0, then the user must apply strategy GoUp.

The If strategy does away with the need for several grammar rules to describe a conditional strategy. The If strategy is defined as:

```
If[$condition, $s1, $s2]:=Code[If[$condition[[1]]]===True, $Failure]], $s2]

If[$condition, $s1, $s2]:=Code[If[$condition[[1]]]=!=True, $Failure]], $s1]
```

The first line states that If[condition, $s1, $s2] can be applied if the condition is not True (because then the code block returns $Failure) and $s2 can be applied. The second line states that If[condition, $s1, $s2] can be applied if the condition is not False (because then the code block would return $Failure) and $s1 can be applied. So, if the condition is True, then $s1 must be applicable (or the whole strategy will fail), and if the condition is False, then $2 must be applicable (or the whole strategy will fail). If the condition is neither True nor False, then also $s2 should be applicable, or the whole strategy will fail. In other words, when an exercise specifies this strategy, the user must apply $s1 if the condition is True, or $2 otherwise.

There is also a shorter form of the If strategy, where it is not explicitly specified what to do if condition is is False:

```
If[$condition, $s] := If[$condition, $s, Pass]
```

While

While[$condition, $s] applies strategy $s while $condition is True.
Attribute $condition must be a Code block (or any other holded expression, e.g. Hold or HoldComplete).

The $condition is typically a condition on the $CurrentTerm, modified by a rewrite rule. For example, consider:

```
add1: x-> x+1
DoAdd1: {add1}
T:=While[Hold[$CurrenTerm < 5], DoAdd1]
```

Given `$CurrentTerm=1`, `T` is applicable if `{add1, add1, add1, add1}` has been applied.

The While strategy is defined as follows:

```
While[$condition, $s] := RepExh[If[$condition, $s, Fail]]
```

ForSeq
ForAll[$var, $values, $s] repeatedly applies $s for all values in $values, assigned to $var, in sequential order. Here, $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

ForAll
ForAll[$var, $values, $s] applies $s for all values in $values, assigned to $var, in any order order. Here, $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

ForOne
ForOne[$var, $values, $s]: applies strategy $s for any one (and only one!) value in $values, assigned to $var. Here $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

Pass
The strategy Pass applies no rewrite rules but never fails. This is sometimes handy to make a strategy more readable.

Fail
The strategy Fail always fails, i.e. the opposite of Pass. In fact, it is defined as: Fail:=not(Pass).

Note: For technical reasons (this is a limitation of the current parser implementation) any sub-strategies called by the constructions above (i.e. the strategies assigned to $s, $s1, $2 etc) cannot refer to rewrite rules. Instead, the attribute should be a simple strategy that applies the rewrite rule, e.g. instead of `While[Code[...], makezero]` we should write `While[Code[...], MakeZero]` and `MakeZero:=makezero`.

Example strategy: SolveLinearEqns

In this section a strategy is presented for solving a system of m linear equations in n variables, where the coefficient for each variable must be a numerical value (no parameters are allowed in the equation). The strategy is basically to first rewrite the set of equations to a matrix, which is then row-reduced using Gaussian elimination, and finally converted back to a set of equations, which should then be trivial and is considered the solution to the system.

Note that Gaussian elimination is a strict procedure; there is only one correct sequence of rewrite steps that satisfies this strategy. It is not really a desirable strategy for use in the LA system, because the user will be forced to apply rules in a fixed order even it is obvious that some steps are independent and can be applied in any order.

Formula terms

The initial formula that is given in the exercise is a `LAEqnSet`. This term should contain a list of equations, each equation constructed by nested `LAPlus` and `LATimes` terms. At deepest level in the expression are either numbers or `RealVar` terms.

The `LAEqnSet` should be rewritten to an `AugmentedMatrix`. The `AugmentedMatrix` term contains an `LAMatrix` term and a list of variables (`RealVar` terms).

The strategy for Guassian elimination actually works on the `LAMatrix` term (not on the `AugmentedMatrix` term). The `LAMatrix` term consists of a list of rows and each row is a list of numbers. To retrieve a row `$i` from a matrix `$mat`, in a Code block, we write: `$row = $mat[[$i]]`. To retrieve an element of the matrix at row `$i`, column `$j`, we write: `$element = $mat[[$i, $j]]` or `$row[[$j]]`.

Rewrite rules

To solve a system of linear equations, we need the following rewrite rules:

`eqns2aug[$solvevars]`: convert a set of equations to an augmented matrix, where $solvevars is the list of variables associated with the matrix.

`aug2eqns`: convert an augmented matrix to a set of equations

`swaprows[$r1,$r]`: swap rows $r1 and $r2 in a matrix

`mulrow[$r, $k]`: multiply row $r in a matrix with factor $k

`muladdrow[$r1, $r2, $k]`: multiply row $r1 in a matrix with a factor $k and add the result to row $r2.

Grammar rules

The start term for the strategy is `SolveLinearEqs[$eqnpos,$solvevars]`. Here `$eqnpos` is the position of the equation set in the exercise term that should be solved, and `$solvevars` is a list of the variable names for which the system is to be solved.

To apply the `SolveLinearEqns` strategy, the user must first transform a system of equations to an augmented matrix (rule `eqns2aug`). The variables in which the system is to be solved (`$solvevars`) are used as the variables above the matrix. Next, the matrix should be transformed to reduced echelon form (`GaussElim`) and finally the system is converted back to a system of equations, which will be a set of trivial equations if the matrix could be completely reduced.

```
SolveLinearEqns[$eqnspos,$solvevars]:=
      eqns2aug[$solvevars]
      (* matrix is one level deeper than result of eqns2aug *)
      Code[$matpos = Append[$eqnspos, 1]]
      GuassElim[$matpos]
      aug2eqns
```

`GuassElim` consists of a forward pass and a backward pass.

```
GaussElim[$matpos]:=
      ForwardPass[$matpos],
      BackwardPass[$matpos]
```

The forward pass repeats a number of steps (`ForwardSteps`), and for each step one more row is covered up (`$coveredrows`) at the top of the matrix. Each iteration will create a column with a pivot with value 1 in the top row (the first un-covered row) and zeroes below it.

```
ForwardPass[$matpos] :=
      Code[
            $mat=Extract[$CurrentTerm, $matpos];
            $range=Range[0, Length[$mat]-1]
      ]
      ForAll[$coveredrows,$range,ForwardSteps,$matpos,$coveredrows]]

ForwardSteps[$matpos,$coveredrows] :=
      FindColumnJ[$matpos,$coveredrows,$j],
      ExchangeNonZero[$matpos,$coveredrows,$j],
      ScaleToOne[$matpos,$coveredrows,$j],
      MakeZeroesFP[$matpos,$coveredrows,$j]
```

`FindColumnJ` determines the value of `$j` such that column `$j` is the first column with a non-zero value in it, or fails if no such column exists.

```
FindColumnJ[$matpos, $coveredrows,$j]:=
      Code[ $mat=Extract[$CurrentTerm, $matpos];
            If[$coveredrows>=Length[$mat],$Failure,
                  $m=Transpose[Drop[List@@$mat,$coveredrows]];
                  $j=SelectIndex[$m,!zerovector[#]&];
                  If[$j==={},$Failure,$j=$j[[1]]]
            ]
      ]
```

If column `$j` has a zero in the first uncovered row, `ExchangeNonZero` swaps that row (using rewrite rule `swaprows`) with the next row that has a non-zero entry, if possible. In the implementation, if entry in the first uncovered row is non-zero, then `$row1==$row2` and no swap is applied.

```
ExchangeNonZero[$matpos,$coveredrows, $j]:=
      Code[
            $mat=Extract[$CurrentTerm, $matpos];
            $cols=Transpose[Drop[List@@$mat,$coveredrows]];
            $row1=$coveredrows+SelectIndex[$cols[[$j]],#=!=0&][[1]];
            $row2=$coveredrows+1
      ]
      If[Code[$row1!=$row2],swaprows[$row1,$row2]]
```

`ScaleToOne` scales the pivot (the entry in the first uncovered row in column `$j`) to one if necessary, using the rewrite rule `mulrow`.

```
ScaleToOne[$matpos,$coveredrows,$j]:=
      Code[
            $m=Transpose[Drop[List@@$mat,$coveredrows]];
            $firstuncoveredrow=$coveredrows+1
            $k = $mat[[$firstuncoveredrow,$j]];
      ],
      If[Code[$k!=1 && $k!=0], mulrow[$firstuncoveredrow,$k]]
```

`MakeZeroesFP` (FP stands for forward pass) makes zeroes in column `$j` by adding a multiple of the first uncovered row to the rows below it.

```
MakeZeroesFP[$matpos,$coveredrows,$j] :=
      Code[ $mat = Extract[$CurrentTerm, $matpos];
            $pivotrow = $coveredrows+1;
            $range = Range[$pivotrow+1, Length[$mat]-1]
      ]
      ForAll[$row, $range, MakeZero[$matpos, $pivotrow, $row, $j]]
```

`MakeZero` makes a zero in column `$j` of row `$torow`, by adding `$k` times row `$fromrow`, where `$k` is the value in row `$torow`, column $j (`$mat[[$torow, $j]]`). Only if $k != 0, then rewrite rule `muladdrow` is applied.

```
MakeZero[$matpos, $fromrow, $torow, $j]:=
      Code[ $k = -$mat[[$torow,$j]] ]
      If[Code[$k != 0], muladdrow[$fromrow,$torow,$k]]
```

The backwards pass works from the bottom up. For each row (`$fromrow`) with a pivot (implemented by the `Select` function), we make zeroes above the pivot, using sub-strategy `MakeZeroesBP`.

```
BackwardPass[$matpos]:=
      Code[$mat = Extract[$CurrentTerm, $matpos];
           $range = Select[Range[Length[$mat],2,-1],
                  Not[zerovector[$mat[[#]]]&
            ];
      ]
      ForAll[$fromrow, $range, MakeZeroesBP[$matpos,$fromrow]]
```

`MakeZeroesBP` (BP stands for backwards pass) makes zeroes above a given row (`$fromrow`) in the pivot column (`$j` is the first column with a non-zero entry in row `$fromrow`). `MakeZeroesBP` makes use of `MakeZeroes`, defined earlier.

```
(* make zeroes above row $fromrow *)
MakeZeroesBP[$matpos,$fromrow]:=
      Code[ $mat = Extract[$CurrentTerm, $matpos];
            $range = Range[$fromrow-1,1,-1];
            $j = SelectIndex[$mat[[$fromrow]],notzero[#]&];
      ],
      ForAll[$torow, $range, MakeZero[$matpos,$fromrow, $torow, $j]]
```

8. Feedback

The LAsystem can give feedback to students making an exercise, based on their progress. The actions of the student are compared to the strategy defined for the exercise and the following situations can occur:

- the student is on track: no feedback necessary.
- the student is not on track: the system will undo the last step.
- the student has finished the exercise. The student will be informed and no more rewrite rules can be applied.

Also, the student can explicitly ask for a help:

- The student asks for a hint. Each time the student ask for a hint, a progressively more specific hint will be given.
- The student ask the system to do one step. This can be repeated until the whole exercise is finished.
- The student ask the system for the complete solution to the exercise.

For each gterm (grammar rule or rewrite rule) a hint can be programmed. The first hint that is given by the system is the hint associated with the highest level sub-strategy, leading to the next to-be-applied rewrite rule. A subsequent hint will be the hint associated with a highest level sub-strategy below the previous one. The last hint that can be given, and will be repeated if more hints are requested, is the hint associated with the lowest level sub-strategy, just above the next rewrite-rule-application.

Adding hints

Hints can be programmed alongside a strategy, by defining extra patterns for the **`Hint`** function, as follows:

```
Hint[gterm[name, attributes...]] := “text”
```

If the user must visit some sub-strategy, matching the attributed name in the definition above, then the hint text, given on the right-hand side of the definition, can be given if a hint is requested.

The name and attributes should, in most cases, be Mathematica patterns (using blanks etc, see Section 6), such that the hint will match for any attribute values. Pattern variables can be used in the hint text also, for example:

```
Hint[gterm["Strat25c", matpos_, coveredrows_, j_]] :=
"Create zeroes under the pivot in column " <> ToString[j] <>
" by adding multiples of the pivot row to other rows."
```

In general, hints should describe a sub-strategy in general terms, for example, the type of operations needed, and the goal of the sub-strategy.

Appendix A: formula terms

This appendix describes the basic Mathematica symbols that represent LA system terms. For more information and functions to manipulate terms, see `LAterms.nb`.

`AugmentedMatrix[_LAMatrix, _List]`

represents a augmented matrix, i.e. a matrix with an associated set of variables printed above it, equivalent to a system of linear equations in those variables. The matrix is represented by a LAMatrix term. The set of variables is represented by a list.

Example term:

```
AugmentedMatrix[
      LAMatrix[{1, 2, 3}, {4, 5, 6}, {7,8,9}],
      {RealVar[x1], RealVar[x2]}
]
```

Represented formula:

$$\begin{matrix} x_1 & x_2 \\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \end{matrix}$$

`LAEqnSet[___LAInequality]`

represents a set of equations. Each elements of the term should be an LAInequality, representing an equation.

Example term:

```
LAEqnSet[
      LAInequality[LAPlus[RealVar[x1], LATimes[5, RealVar[x2]]]], Equal, 1],
      LAInequality[
            LAPlus[
                  LATimes[-2, RealVar[x1]],
                  LATimes[-1, 7, RealVar[x2]]
            ],
            Equal,
            -5
      ]
]
```

Represented formula:

$$\begin{Bmatrix} x_1 + 5x_2 = 1 \\ -2x_1 - 7x_2 = -5 \end{Bmatrix}$$

`LAInequality[lhs_, comp_, rhs_]`

represents an equation `lhs comp rhs`, where `comp_` is one or `Equal`, `Less`, `LessEqual`, `Greater` or

`GreaterEqual`, and `lhs_` and `rhs_` can be any terms.
Example term:

```
LAInequality[LAPlus[RealVar[x1], LATimes[5, RealVar[x2]]]], Equal, 1],
```

Represented formula:

$x_1+5x_2=1$

`LAMatrix[_List]`
represents a matrix, where each element in the list represents a row, and each row is represented by a list of numbers (or other symbols or terms).

Example:

```
LAMatrix[{1, 2, 3}, {4, 5, 6}, {7,8,9}],
```

Represented formula:

$$\begin{matrix} & x_1, x_2 & \\ \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} & & \end{matrix}$$

`LAPlus[a_, b_]`
Represents a+b, where a and b can be any term.

`LATimes[a_,b_]`
Represents a*b, where a and b can be any term.

`LADivide[a_,b_]`
Represents a/b, where a and b can be any term.

`LAPower[a_,b_]`
Represents a^b, where a and b can be any term.

`RealVar["name"]`
Represents a real valued variable, displayed as name. The name argument must be a string. To represent a variable with a subscript, a special notation is used.

example:

```
RealVar["\!\(x\_1\)"]
```

represented formula:

x_1

`RealPar["name"]`
Represents a real valued parameter, displayed as name. The name argument must be a string.
Parameters are treated differently from variables when displaying formulas and and converting between different equation forms.

`LAImaginary`
Represents the imaginary number i (the number i such that i*i=-1)

`LAEmptySet`
Represents the empty set ∅.

Appendix B: rewrite rules

This appendix describes some basic rewrite rules implemented in the LA system. For more information and rewrite rules, see basicrules`.nb`

`eqns2aug[$pos, $usevars]`

Convert set of equations (LAEqnSet) at position $pos to an augmented matrix, separating coefficients of variables $usevars into columns, and using the last column for all constants)

`aug2eqns[$pos]`

Convert augmented matrix (AugmentedMatrix) at $pos to a set of equations.

`swaprows[$matpos, $row1, $row2]`

Swap row number $row1 with row number $row2 in the matrix (LAMatrix) at position $matpos.

`mulrow[$matpos, $row, $k]`

Multiply row number $row with a factor $k in the matrix (LAMatrix) at position $matpos.

`muladdrow[$matpos, $row1, $row2, $k]`

Multiply row number $row1 with a factor $k and add the result to row number $row2 in the matrix (LAMatrix) at position $matpos.

`rowreduce[$pos]`

Row reduce the matix (LAMatrix) at position $pos.

eqns2empty[$pos]

If the set of equations (LAEqnSet) at $pos contains (0=1) or a similarly inconsistent equation, replace it by LAEmptySet.

`var2par[$pos, $var, $par]`

In the equation set (LAEqnSet) at position $pos, add an equation $var=$par, and replace all other occurrences of $var with $par, where $var should be a RealVar and $par should be a RealPar.

eqns2pareq[$pos]

Convert the equation set (LAEqnSet) at position $pos with a paramter representation, i.e. an equivalent single equation (LAInequality), where variables are represented in a single vector on the left-hand-side and all parameters are separated on the right hand side.

delrow[$pos, $row]

In the matrix (LAMatrix) at position $pos, delete row numbered $row.

rowspace[$pos]

Replace the matrix at position $pos with its row-space (LASpan)

colspace[$pos]

Replace the matrix at position $pos with its column-space (LASpan)

addident[$pos]

If the matrix (LAMatrix) at position $pos is NxN, then add the NxN identity matrix on the right to the matrix.

delident[$pos]

If the matrix (LAMatrix) contains the NxN idenity matrix in the first N columns on the left, then remove those N columns.

noinverse[$pos]

Replace the 2NxN matrix (LAMatrix, used for constructing an inverse) at position $pos with the message “matrix has no inverse” .

mateq0[$pos]

Replace the matrix A (LAMatrix) at position $pos with the equation (LAInequality) Ax = 0, where x is a vector (LAMatrix) containing an appropriate number of variables, and 0 is the 0-vector (LAMatrix).

`eq2aug[$pos]`

Convert a single (vector-) equation to an augmented matrix (AugmentedMatrix).

Automatic rules

These rules are special in that it is automatically executed when specified by a strategy, or, if no strategy is specified, then it is automatically executed whenever a term can be simplified. Thus, the user of the LAsystem never explicitly applies these rules, and they are not visible in the list of applicable rules.

simplify[$pos]

Simplify the expression at given position. Simple arithmetic expressions that evaluate to a constant are simplified, and any constant factor 1 before or after a variable/parameter is removed. Attribute $pos is always set to {} when the rule is applied by the LAsystem.

substituteknown[$pos]

Substitutes known parameters in the formula. Parameters are known if there is an equation LAEqual[par, value] in the system, where par is either a RealPar, VectorVar or MatrixPar. If the known parameter occurs on the left-hand side of an equation that contain no variables, then the substitution is not done, so that equations that are used as definitions are not affected. Attribute $pos is always set to {} when the rule is applied by the LAsystem.

distribute[$pos]

Distributes LAUnion over LAEquationSet. So, when an LAUnion occurs in a LAEqnset, the formula is rewritten such that it is a LAUnion of LAEqnsSets. Attribute $pos is always set to {} when the rule is applied by the LAsystem.

Appendix C: strategies

This appendix describes some basic strategies implemented in the LA system. For more information and strategies, see `basicstrat.nb`

High level strategies

`SolveLinearEqns[$pos, $vars]`

Strategy for solving the system of equations (LAEqnSet, LAEqual or AugmentedMatrix) at position $pos for variables in the list $vars (List). The system is solved by converting it to an augmented matrix, row-reducing it and formulating the solution as a parameter representation, if possible.

`GaussEchelon[$pos]`

Strategy for Gaussian elimination, up to echelon form, applied to matrix (LAMatrix) at $pos. Implements part of Strategy 2.5 from Hans Cuypers “Strategies for Linear Algebra”.

`GuassElim[$pos]`

Strategy for Gaussian elimination, to reduced echelon form, applied to matrix (LAMatrix) at $pos. Implements Strategy 2.5 from Hans Cuypers “Strategies for Linear Algebra”.

`RowEchelon[$pos]`

Strategy for reducing a matrix at $pos to echelon form, with more freedom than GuassEchelon

`RowReduce[$pos]`

Strategy for reducing a matrix at $pos to reduced echelon form, with more freedom than GaussElim

Low-level strategies

`RepExh[$s]`
Repeatedly applies the strategy $s, until it fails.

`If[$condition, $s1, $s2]`
Applies strategy $s1 if condition evaluates to True or else applies strategy $s2.
Note: $condition must be a Code block (or any other holded expression, e.g. Hold or HoldComplete).

`While[$condition, $s]`

Applies strategy $s while $condition is True.
Attribute $condition must be a Code block (or any other holded expression, e.g. Hold or HoldComplete).

`ForSeq[$var, $values, $s]`

Repeatedly applies $s for all values in $values, assigned to $var, in sequential order. Here, $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

`ForAll[$var, $values, $s]`

Applies $s for all values in $values, assigned to $var, in any order. Here, $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

`ForOne[$var, $values, $s]`

ForOne[$var, $values, $s]: applies strategy $s for any one (and only one!) value in $values, assigned to $var. Here $var is a $-variable, $values is a list of values for $var and $s is a strategy using $var as an attribute.

`Pass`

Can always be applied successfully. Sometimes useful as attribute to other strategy.

`Fail`

Can never be applied (always fails) . Sometimes useful as attribute to other strategy.

TryAutomatricRules

Will try to apply the automatic rules “simplify”, “substituteknown” and “ distribute”.