

# logisolve — A logic-puzzle solver in ASF+SDF

A.T. Hofkamp

Id: doc.tex,v 1.4 2004/08/13 13:51:24 hat Exp

## Abstract

While the meta manual ([vdBK03]) does a good job of explaining all the facilities available in meta, it does not tell you how to write an application in ASF+SDF. This document aims to close that gap (at least partly) by starting where the manual ends. It gives an example of how an problem (in this case a logic-puzzle solver) can be solved using the language, and explains a number of useful conventions along the way.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Logic puzzles . . . . .	2
1.1.1	The solving algorithm . . . . .	2
1.2	Assumed knowledge . . . . .	4
1.3	How to read this document . . . . .	4
<b>2</b>	<b>Cycle 1: Syntax definition</b>	<b>4</b>
2.1	Input grammar . . . . .	5
<b>3</b>	<b>Cycle 2: Conversion to AST</b>	<b>8</b>
3.1	AST format definition . . . . .	8
3.2	Input to AST transformation . . . . .	10
<b>4</b>	<b>Cycle 3: Semantic checking</b>	<b>14</b>
4.1	The output format . . . . .	14
4.2	Checking values in categories . . . . .	16
4.3	Checking use of identifiers in expressions . . . . .	16
<b>5</b>	<b>Cycle 4: Deducing variables</b>	<b>17</b>
5.1	Variable table . . . . .	18
5.2	Finding variable values . . . . .	18
<b>6</b>	<b>Cycle 5: Simplifying expressions</b>	<b>20</b>
6.1	Filling in variables . . . . .	20
6.2	Simplifications (part 1) . . . . .	21
6.3	Simplifications (part 2) . . . . .	22
<b>7</b>	<b>Cycle 6: Gluing everything together</b>	<b>22</b>
7.1	Writing the solve-problem function . . . . .	23
<b>8</b>	<b>Cycle 7: Reasoning expressions</b>	<b>25</b>
8.1	Introducing the Relation sort . . . . .	26
8.2	Generating reasoning expressions . . . . .	27

<b>9</b>	<b>Cycle 8: Output format</b>	<b>29</b>
9.1	Making a compact output format . . . . .	29
<b>10</b>	<b>Next steps</b>	<b>32</b>
<b>11</b>	<b>History</b>	<b>32</b>

# 1 Introduction

After writing a compiler using ASF+SDF, I reflected how the project went. One of the conclusions that I drew was that the formalism could use an ‘application manual’, a manual that explains how to use all the features of the language. The current user manual [vdBK03] only explains the available features, it does not explain how to combine them and put them to good use. A few weeks later I saw an opportunity to fix this problem (at least partly, I hope). I often ponder about algorithms. Every now and then a new approach takes shape in my head, which I then want to try by implementing the algorithm. Putting both together, the logisolve program and this document were born. Note that I wrote the program and this paper to satisfy my own curiosity, I do not get paid to do this. I do believe that the ASF+SDF formalism has a place in the universe, I also believe that it will never be ‘the solution to all problems’.

In this paper, the road to the final solution is at least as important as the implementation itself. The development method used here is incremental extension, that is, the program is implemented by repeatedly extending the program with new or updated functionality. Such development techniques are commonly known as Rapid Application Development (RAD), employed for example in eXtreme Programming (XP). Each cycle of development is described in a separate section. In each section, the next goal is explained, exercises to reach the goal are presented, and finally my solution to reach the goal is discussed. In the discussion, the application code is only globally explained, the more interesting part of the discussion is about why this solution was chosen with respect to the ASF+SDF formalism. You are free to disagree with my solution or with my reasons, there is always more than one good solution to a problem.

While the application is quite small, it contains many of the elements of a typical compiler program. First the input is parsed (Cycle 1), then a conversion to abstract syntax tree (AST) is performed (Cycle 2), followed by semantic checks (Cycle 3) and some processing (Cycles 4–7). Finally output is generated (Cycle 8), followed by some ideas how to continue with the development of this program, and an overview of how the document came to its current form (for the curious).

## 1.1 Logic puzzles

The goal of the program is to solve logic puzzles that often appear in magazines. An example of such a puzzle is shown in Figure 1. It is also used as test problem in the paper. No doubt you can solve this puzzle without using the program. Feel free to use a different (more difficult) example, although there is no guarantee that the program will be able to solve it using the algorithm explained below (but that is a good excuse to further expand the program).

### 1.1.1 The solving algorithm

The algorithm to solve these puzzles basically uses reasoning with boolean expressions. The puzzle description introduces categories, sets of identifiers associated with the category. For simplicity, it is assumed that each identifier is unique (that is, it is not allowed to have the same identifier in more than one category). The output is a list of sets related identifiers. For example, if Jaap was in Paris drinking wine is a solution (which it is not), the list would contain the set {Jaap, Paris, wine}.

Four people sit down in a cafe in a famous city, and order their favorite drink. Can you find out who is drinking what and where?

<b>Category</b>	<b>Values</b>
People	Jaap, John, Jacob, and Jeen
City	Paris, London, Brussels, and Berlin
Drink	Coffee, Tea, Beer, and Wine

Hints:

1. Jaap is drinking neither tea nor coffee.
2. Jeen is sipping her wine.
3. Tea is served either in Brussels or in Berlin.
4. Beer is served in Berlin.
5. John enjoys his drink below the Eiffel tower.

Answer by completing the following table:

<b>Person</b>	<b>City</b>	<b>Drink</b>
Jaap		
John		
Jacob		
Jeen		

Figure 1: Logic-puzzle example.

The output is computed by deciding for each pair of identifiers whether they belong together or not. Such a pair is effectively a boolean variable. For example ‘Jaap/Paris’ is such a variable. If the variable is true, identifiers ‘Jaap’ and ‘Paris’ are related, if the variable is false, the identifiers are not related. Assigning values cannot be done randomly, there are restrictions that must be obeyed. These restrictions take the form of boolean expressions that have to evaluate to true after filling in all variables.

The program uses the restricting expressions to reason about the value of variables. For example, the second hint gives the expression ‘Jeen/Wine’. The only way to make this expression evaluate to true is to assign true to the Jeen/Wine variable. This information is then used to simplify other expressions, which in turn delivers more information (hopefully), etc, until all variables are assigned a value, in which case we are finished.

## 1.2 Assumed knowledge

It is assumed that you have knowledge about programming, in particular functional programming, since ASF+SDF is essentially a functional programming language with user-defined syntax. Some global knowledge about compiler design may also be helpful. Also, you are expected to have read (and understood for the most part) the user manual ([vdBK03]). This paper starts where the manual ends (namely, how to actually write an application using ASF+SDF). Finally, you should have meta installed (the ASF+SDF tool set) at your computer, so you can try everything yourself. There is no substitute for learning by doing!

## 1.3 How to read this document

Each of the following sections discusses a single development cycle. The first part of each section explains the next goal that should be reached. The idea is that you try to do the same cycle while you read. To guide you in the process, the second part of the section consists of one or more ‘exercises’ that together give you the solution. The third and last part of each section is also the biggest. It globally discusses the solution I wrote, and goes into more detail with specific ASF+SDF aspects of the code by showing fragments. In particular, it gives you conventions to use when writing your own programs using ASF+SDF. The biggest advantage of the language is its adaptability, but without useful conventions, that is also its biggest disadvantage.

As usual with learning programming, trying to solve the problem for yourself some time before looking at a solution helps in understanding the issues involved. It is up to you to decide how much time you want to spend on the exercises before continuing to read my solution and discussion. The minimum I recommend to do is to think how the algorithm that implements the exercises should look like in a functional program (the latter is especially important if you have little previous experience in functional programming), and then check your ideas against my code. The maximum you can do is to solve and code the problem yourself before reading my solution. Although it will be a very good exercise in programming ASF+SDF, it may also cost a lot of time.

For the nitty gritty code details, accompanying this paper is a series of directories `v1` through `v8`, where you can find my resulting code after each cycle.

Last but not least, the conventions I provide are not carved in stone. The idea is mainly to avoid the sea of possibilities that ASF+SDF gives you by offering some beacons. As you grow more accustomed to the language, you can develop your own set of conventions.

Happy sailing!!

## 2 Cycle 1: Syntax definition

First order of business is to define the input syntax, that is, the grammar that allows us to describe the logic puzzle.

```

module tokens

imports
  basic/Whitespace
  basic/Comments

exports
  sorts
    Identifier Variable

  lexical syntax
    [A-Za-z] [A-Za-z0-9]*          -> Identifier
    Identifier "/" Identifier      -> Variable

  lexical restrictions
    Identifier -/- [A-Za-z0-9]

```

Figure 2: v1/tokens.sdf file

### Exercises:

- Write a grammar that accepts a puzzle, containing at least categories and boolean expressions.

## 2.1 Input grammar

The two main parts needed in the syntax are the definition of categories, and the definition of boolean expressions. Additionally, you may want to include a way to enter the hints into the input, and a way to link hints and expressions together, so you can obtain information like ‘give me the expressions that are related to the third hint’.

Since hints are not needed to implement the reasoning algorithm I left them out in my solution, but feel free to include them if you like. Also, I have split the syntax in two parts: a generic part and an input part. Why I did that is explained below. The former is listed in Figure 2, and the latter is listed in Figure 3.

The first file, the `tokens` module, contains basic tokens that are generally usable, not only in the input grammar, but also elsewhere (we will see this happen further in the development of the application). The `basic/Whitespace` import defines the whitespace. This is then extended by the `basic/Comments` import, that introduces line comments starting with ‘%%’ as in ASF+SDF (as well as the ‘%...%’ comment). Next it defines the syntax for an identifier with its lexical restrictions, and the syntax for a variable, also at lexical level, which means that you cannot put white-space in a variable like ‘x /y’ (this text will be recognized as identifier ‘x’, followed by a parse error, because there is nothing in the grammar that starts with a slash-character. On top of the generally usable tokens, the grammar for the input is defined as non-terminal `Problem` in the `logi-syntax` module, that consists of one or more categories, and zero or more expressions. The former are a list of identifiers, and the latter are basic boolean expressions with variables (the syntax of which is defined in the `tokens` file). Besides the usual ‘and’, ‘or’, and ‘not’ operators, equality (‘=’) and implication (‘ $\Rightarrow$ ’) are also included. Note that I used the `constructor` attribute, because I do not want to use this syntax as outermost symbol in an equation. The lines with `reject` at the end prevent the keywords of the language to be interpreted as identifiers (so you cannot have an identifier ‘true’ in a category). For disambiguation of expressions, the association and the relative priorities of the binary operators are also defined. Last but not least a bracket operator is defined.

And that is it. You can try the grammar by entering a term like

```

module logi-syntax

imports
  tokens

exports
  sorts
    Problem Category Expression

  context-free syntax
    Category+ Expression*      -> Problem

    "{" {Identifier ","}+ "}" -> Category

    "true"                      -> Expression {constructor}
    "false"                     -> Expression {constructor}
    "(" Expression ")"          -> Expression {bracket}
    Variable                    -> Expression
    "not" Expression            -> Expression {constructor}
    Expression "=>" Expression -> Expression {nonassoc,constructor}
    Expression "=" Expression  -> Expression {nonassoc,constructor}
    Expression "and" Expression -> Expression {left,constructor}
    Expression "or" Expression -> Expression {left,constructor}

    %% make them proper keywords
    "true"                      -> Identifier {reject}
    "false"                     -> Identifier {reject}
    "not"                        -> Identifier {reject}
    "and"                         -> Identifier {reject}
    "or"                          -> Identifier {reject}

  context-free priorities
  {
    "not" Expression            -> Expression
  } > {
    Expression "=>" Expression -> Expression
  } > {
    Expression "=" Expression  -> Expression
  } > {
    Expression "and" Expression -> Expression
  } > {
    Expression "or" Expression -> Expression
  }

hiddens
  context-free start-symbols
    Problem

```

Figure 3: v1/logi-syntax.sdf file

```
{ jaap, john, jacob, jeen }
{ paris, london, brussel, berlin }
{ coffee, tea, beer, wine }

jeen/wine
```

as input term of the `logi-syntax` module, and check that meta recognizes it as sort `Problem`.

As you can see, I use multiples of four positions for indenting my code. Below the `module` line, I always start with a single `import` section, followed by a single `exports` section. If available, the syntax part of the module ends with a single `hiddens` section. Within the export section, I always first list the new sorts, followed by syntax definitions in the lexical domain (if they exist in the module). Finally, the exported syntax definitions and restrictions in the context-free domain are listed. In modules that define a grammar, the `hiddens` section is normally very small and only contains start symbol definitions. I use empty lines to separate different groups in the same (sub)section. Other people use multiple `exports` and/or `hiddens` sections (one for each group). For example, an exported section for the definition of the `Problem` sort, one for the `Category` sort, and one for the `Expression` sort.

For module names, I use lowercase names for ‘normal’ modules, and names with an initial uppercase letter for parameterized modules. Longer names are made readable by inserting dashes in the name as in `logi-syntax`. Other people use module names like `LogiSyntax`. However I like having lowercase file names (remember, module names are also used as file name), so I do not like that convention.

With respect to contents of modules, I have modules that define syntax (for example the `tokens` and the `logi-syntax` modules shown here), and modules that define functions (or parts of functions). We will encounter the latter type in the next development cycles. I rarely combine syntax and functions in one module. To denote that a module contains syntax, I use a `-syntax` suffix in the module name. For a small application, this should be sufficient. For bigger applications, you may want to split one grammar over multiple modules, and put them all in the same directory, so you get module names like `user-syntax/expressions`.

In the same way, it is also a good idea to consider naming conventions of sort names. `ASF+SDF` requires an initial uppercase letter, followed by letters, digits or dashes. Instead of having a very generic sort name like `Expression`, you may want to have a common prefix for sorts in the same grammar, for example `User-expression`. In that way, you avoid name clashes.

In the production rules, I always write double quotes around literals. Also, I line up the arrows so they appear underneath each other. In this way, it is easy to find sets of related rules that all give the same result sort. Lining up arrows is easiest if you put the ‘-’ at a tab stop (a multiple of four positions in my case). Last but not least, I have configured my editor to understand the conventions I use, and do proper syntax highlighting.

In the `priorities` (sub)section, as you can see, I write the priority group separator ‘} > {’ on a separate line, left-aligned with the ‘c’ of `context-free`. Rules in the same priority group are written directly underneath each other. This convention gives you clear visual clues which priority groups exist. It also gives you space to write properties of each group (not shown here), which go between the opening curly bracket and the first production rule of the group. Last but not least, if you are careful, this convention allows for easy construction of the priorities section. While writing the syntax rules of the grammar, think about the priorities of the operators. Make sure that you order the rules from high to low. (This is something you will want to do anyway for a readable grammar.) The priorities section can be made by copying the rules from the syntax definitions, stripping away the annotations (you can also leave them in, but then you need to keep the annotations in sync with the syntax definition), deleting the irrelevant lines, and inserting one ‘} > {’ line. Yank that line, and paste it at the other points between two priority groups. Finally, add the first and last curly brackets. (The alternative to stripping away annotations is to make a copy of the production rules

before adding annotations to the syntax rules.) The `start-symbols` section is always hidden. This prevents propagation of start symbols upwards, and as a result it minimizes the number of start symbols in each module. That gives smaller parse tables and faster compilations.

The only remaining ‘strange’ thing here that I have not explained is that two files are used to define one syntax even for this small grammar. The reason for splitting the grammar is that further in the development, we will define additional grammars in the application, and we want to share the common part, because it makes transformation from one grammar to another trivial. (Without common part, we must break down the tree to the lexical level using the lexical constructor functions of ASF+SDF and then reconstruct a tree of the appropriate sort in the new grammar.)

### 3 Cycle 2: Conversion to AST

After having defined the input syntax, the next step is to prepare for processing the data entered. To simplify processing, an internal AST format should be introduced. The second goal is therefore to introduce an AST format into the application. In addition, we need to have a function that rewrites input to the AST format.

#### Exercises:

- Define an AST format by writing a grammar for it.
- Write a conversion function from the input grammar to the AST grammar.

#### 3.1 AST format definition

In ASF+SDF, everything that you can write down needs to be covered by a non-terminal of some sort. That holds for input data (defined in the previous cycle), any output generated by the program (we will see that later), and also for intermediate data structures like the AST.

Defining an AST format is much the same as defining the input format in the previous cycle. However, the AST is considered to be an internal format, which means that the focus for this format is more oriented towards easy processing the data rather than good readability or compactness. In addition, you may want to plan for export of this format into some other tool, perhaps written in a different language. To make that easy, we need an easy-to-parse format. Using a prefix notation covers these requirements. For example, instead of writing an expression like `not true and false`, we write it as `and(not(true()),false())`. This format is easy to parse (an LL(1) parser would be enough), it is unambiguous (the infix version could also be interpreted as `not(and(true()),false())` if you disagree on the priorities between ‘and’ and ‘not’), and a machine can easily build a tree out of the format. Also, you can easily transform this AST format to other syntax, for example XML.

The AST definition that I invented is shown in Figure 4. As you can see it is largely the same approach as the input grammar. The `tokens` module is imported here, so all the definitions from that module are available here as well. There are also a few differences. Firstly, syntactic sugar like the curly brackets around the categories and the comma between identifiers has been dropped. The reason is that we aim for an easy format for a computer program rather than a human being. Secondly, all sort names have an `Ast-` prefix, except for the common `Identifier` sort. This has two benefits. The first benefit is that the grammar of the AST is separate from the input grammar. More about this later, after the transformation function is defined. The second benefit is that it is clear to us (anywhere in the entire program) that some term of sort `Ast-...` is part of the AST grammar, which narrows down the search for its definition. Thirdly, since the format is unambiguous, there is no need to define priorities. The fourth and final difference is the relaxed syntactical restrictions of allowing zero categories and allowing zero identifiers in a category. The idea behind it is that the input grammar already enforced these restrictions, so there is no reason to enforce it in the AST format as well. Also, programming the transformation functions becomes easier in this way.

```

module ast-syntax

imports
  tokens

exports
  sorts
    Ast-problem Ast-category Ast-expression
    Ast-variable

  context-free syntax
    "prob" "(" cats:Ast-category* "," exprs:Ast-expression* ")"
        -> Ast-problem {cons("probl")}

    "cat" "(" idens:Identifier* ")"
        -> Ast-category {cons("cat")}

    "var" "(" iden1:Identifier "," iden2:Identifier ")"
        -> Ast-variable {cons("var")}

    "true" "(" ")"
        -> Ast-expression {cons("true")}
    "false" "(" ")"
        -> Ast-expression {cons("false")}
    Ast-variable
        -> Ast-expression
    "not" "(" expr:Ast-expression ")"
        -> Ast-expression {cons("not")}
    "implies" "(" lft:Ast-expression
        "," rgt:Ast-expression ")"
        -> Ast-expression {cons("impl")}
    "equal" "(" lft:Ast-expression
        "," rgt:Ast-expression ")"
        -> Ast-expression {cons("eql")}
    "and" "(" lft:Ast-expression
        "," rgt:Ast-expression ")"
        -> Ast-expression {cons("and")}
    "or" "(" lft:Ast-expression
        "," rgt:Ast-expression ")"
        -> Ast-expression {cons("or")}

```

Figure 4: v2/ast-syntax.sdf file

```

module convert

imports
  logi-syntax
  ast-syntax-api

exports
  context-free syntax
    "convert" ( Problem ) -> Ast-problem

hiddens
  context-free syntax
    "convert-categories" ( Category* )      -> Ast-category*
    "convert-idents" ( {Identifier ","}+ ) -> Identifier+
    "convert-exprs" ( Expression* )        -> Ast-expression*
    "convert-expression" ( Expression )    -> Ast-expression

  context-free start-symbols
    Ast-problem

hiddens
  variables
    "$cat" [0-9]? -> Category
    "$*cat" [0-9]? -> Category*
    "$+cat" [0-9]? -> Category+
    "$expr" [0-9]? -> Expression
    "$*expr" [0-9]? -> Expression*
    "$iden" [0-9]? -> Identifier
    %% "$*iden" [0-9]? -> Identifier*
    "$idenlst" [0-9]? -> {Identifier ","}+
    "$var" [0-9]? -> Variable

    "$aexpr" [0-9]? -> Ast-expression

    "$+char" [0-9]? -> CHAR+

```

Figure 5: v2/convert.sdf file

The downside of this decision is that it becomes possible to have logic-puzzle problems without any category or without any values in a category, at least in the AST format. This may cause havoc with some applications. If you think that is bad, add the restrictions in the AST format, and ASF+SDF will help you to enforce the restrictions.

As you can see, the syntax definition also has hooks for the ASF+SDF API generator. The generated API is put in the `ast-syntax-api` module (the generator is currently not explained in the user manual).

### 3.2 Input to AST transformation

After defining the AST format, the next step is to define the transformation from input to AST. This is done in `v2/convert` module, shown in Figures 5 and 6 respectively.

This module imports both the input grammar and the API of the AST format. For the transformation, it defines one transformation function `convert`, which takes a value of the sort `Problem`, and outputs a (corresponding) value of the sort `Ast-problem`. Since this is intended to be a function that gets rewritten to its output value, `convert` is a prefix function.

The `convert` function needs a number of sub-functions. Since it makes no sense to expose these to the outside world, they are defined in the `hiddens` section. The local start symbol allows calls to the `convert` function to be parsed (that is, you can process a term of the form `convert(...)`).

As in longer sort names, I also use dashes to make longer function names more readable, and I don't use uppercase letters in it. Like most other things in ASF+SDF, this is a just convention. If you don't like it, pick another one. To make life easier, you will probably want something that looks different than a sort name (although ASF+SDF would allow it).

A new section in the module is the variables section. Here you define the variables that you use in the equations (in `v2/convert.asf`). In principle, you are free to chose any piece of text as variable (for example, the line `"abc" -> Category` would give me a variable 'abc', that I can use as place holder for a category). However, life is easiest if you ensure that variables are written differently than anything else that you define in the grammars involved. Here, starting with the otherwise unused character `$` ensures that you (and the parser) will not get confused. After the `$` character, you need a sequence that uniquely identifies what values you can put into it. I use a short identifier (for example 'expr' means that it can have values of the sort `Expression`, etc) for this purpose. For equivalent concepts in different grammars I use a one letter prefix (for example the sort `Ast-expression` gets the sequence 'aexpr'). A prefix `*` or `+` is added when the variable can hold zero or more, or one or more values respectively. Another variant that sometimes occurs are lists of sorts separated by some character, like `{|identifier ","}*.` I use the suffix 'list' or 'lst' for these values. Finally, to allow the same variable name to be used more than once in an equation, I add an optional single digits. Others use zero or more digits as suffix, or allow other characters such as the single quote in the suffix.

To make it easier to find variables in a long list, I order them. I always keep variables with the same (base)sort together. For example, '\$expr' and '\$\*expr' belong together. Beyond that, I keep variables from different grammars separated by inserting an empty line between different groups. Within the same group, I try to keep the variables sorted, but that does not always succeeds.

Although you can export variables, you are advised not to do that, because it can become very confusing when you ever need to find a variable definition again. Also, meta gives warnings about exported variables. Instead, define hidden variable sections in each module (using the same name for the variable in each module to prevent you from going insane).

In the equation part (in `convert.asf`), you notice that I indent the equation lines four positions. I also maintain a function heading above the rewrite rules of a function. At the start, I write whether it is an internal or an external function. For more complex functions, I often write down some explanation what the function is supposed to do. I find that these comments help to navigate in the file, especially when you configure your editor to do syntax highlighting on comments. The downside of including the information is that ASF+SDF considers it comments, which means that the information is not checked against the real definition, and as a result it may lie to you.

In larger files, you may get groups of functions that you want to keep together. I separate such groups from each other by inserting one or two comment lines '=' characters (in this application, I have not used this convention).

Each equation in the file gets a unique name by using the first two or three letters of each word of the function name, followed by a dash and a number. I do try to keep the letter sequences unique in each file, but that does not always works nicely. In that case, I make the names unique by using unique numbers (that is, continue incrementing the number instead of restarting with number one). If that also fails, I normally give up (this happens very rarely). Keeping the names unique means that you can search for the name of an equation when meta reports an error with an equation-name. Also, you can use the names when referring to equations in email and other documents.

ASF+SDF supports three versions of equation syntax. After some experimenting, I found the 'when' version of equations most readable. Other people use other versions, your favorite version probably depends on your background and taste. I format the equation always such that lines which are too long are split into two or more lines, with the second and further lines are formatted more

```

equations
%%
%% export: convert(Problem) -> Ast-problem
%%
[cv-1] convert($+cat $*expr)=
        make-probl(convert-categories($+cat),
                convert-exprs($*expr))

%%
%% hidden: convert-categories(Category*) -> Ast-category*
%%
[cc-1] convert-categories()=

[cc-2] convert-categories({$idenlst} $*cat)=
        make-cat(convert-idents($idenlst))
        convert-categories($*cat)

%%
%% hidden: convert-idents({Identifier ",")+} -> Identifier+
%%
[ci-1] convert-idents($iden)=$iden

[ci-2] convert-idents($iden,$idenlst)=$iden convert-idents($idenlst)

%%
%% hidden: convert-exprs(Expression*) -> Ast-expression*
%%
[ces-1] convert-exprs()=

[ces-2] convert-exprs($expr $*expr)=convert-expression($expr)
        convert-exprs($*expr)

%%
%% hidden: convert-expression(Expression) -> Ast-expression
%%
[ce-1] convert-expression(true)=make-true()
[ce-2] convert-expression(false)=make-false()
[ce-3] convert-expression($var)=make-var($iden1,$iden2)
        when variable($+char1 "/" $+char2):=$var,
        $iden1:=identifier($+char1),
        $iden2:=identifier($+char2)
[ce-4] convert-expression(not $expr)=make-not(convert-expression($expr))
[ce-5] convert-expression($expr1 => $expr2)=make-impl($aexpr1,$aexpr2)
        when $aexpr1:=convert-expression($expr1),
        $aexpr2:=convert-expression($expr2)
[ce-6] convert-expression($expr1=$expr2)=make-eql($aexpr1,$aexpr2)
        when $aexpr1:=convert-expression($expr1),
        $aexpr2:=convert-expression($expr2)
[ce-7] convert-expression($expr1 and $expr2)=make-and($aexpr1,$aexpr2)
        when $aexpr1:=convert-expression($expr1),
        $aexpr2:=convert-expression($expr2)
[ce-8] convert-expression($expr1 or $expr2)=make-or($aexpr1,$aexpr2)
        when $aexpr1:=convert-expression($expr1),
        $aexpr2:=convert-expression($expr2)

```

Figure 6: v2/convert.asf file

to the right. The ‘when’ is always formatted such that the ‘n’ is below the ‘]’ of the equation name. All conditions are written on separate lines, with the comma separating two conditions written at the end of the line.

With multiple equations for the same function, I always start with the rule that ends the recursion, followed by the non-terminating rules. If there is a default rule (which happens very infrequently, you will see examples in future development cycles), that rule is always last. I normally put an empty line between each pair of equations. The only exception that I make to this rule is with cases like shown here. The rewrite rules of the *convert-expression* function are very similar and easy, so there is no real need to add empty lines. Keep in mind that a variable in ASF+SDF behaves as a variable in the mathematical sense, you can assign a value only once to it, all further uses are then equivalent to replacing the variable by its contents. For example `convert-expression($expr1 and $expr1)` will only match if the left-hand side and the right-hand side of the ‘and’ are equal.

The only equation that may need some additional explanation is [ce-3] (see, using equation names works!). In that rewrite rule, a variable is converted to AST format. The ‘trick’ shown here is quite uncommon (if you are careful). The problem that occurs here is that the Variable sort is defined in the lexical domain instead of the context-free domain. For this reason, you cannot match its value to a pattern like `$iden1/$iden2`, because patterns are always context-free. Instead, you need to access the data at lexical level using the lexically, you need the lexical constructor function *variable* to get its contents. Here, the contents is split to a sequence of CHARs before the slash (in ‘\$+char1’), the slash character itself, and a different sequence of CHARs behind the slash (‘\$+char2’). After the lexical pattern match, you have the ‘\$+char1’ and ‘\$+char2’ variables containing sequences of characters. Using the *identifier* lexical constructor function, both sequences are converted ‘normal’ identifiers (that is, values of sort Identifier), and put into the Ast-variable sort.

Finally, I promised to explain why we need new sort names for the AST. The other solution would be to use the same sort names in both grammars. This would merge the AST syntax with the syntax of the input grammar. That means the user can use internal AST notation as input and vice versa. Clearly, this is unwanted behavior.

For testing, try to parse and reduce an input term like

```

convert(
  { jaap, john, jacob, jeen }
  { paris, london, brussel, berlin }
  { coffee, tea, beer, wine }

  not jaap/tea
  not jaap/coffee
  jeen/wine
)

```

The output you will get is something like ‘`prob(cat(jaap john jacob jeen) cat(paris london brussel berlin) cat(coffee tea beer wine), not(var(jaap, tea)) not(var(jaap, coffee)) var(jeen , wine))`’. You may think (correctly, in my opinion) that this is not very readable. However, keep in mind that *a*) the output is in AST format, which is not designed to be human readable, *b*) meta ensures that output is compliant with the syntax definition of the grammar (here, the Ast-problem sort), so output is less interesting than you may think at first, *c*) ASF+SDF assumes that context-free grammars are used, for which layout is not important, *d*) there are people busy to get some form of pretty printing working in meta which means that the problem should disappear in the future, and *e*) if you want pretty-printed output, you will need to invest additional time to define what ‘pretty’ means. For now, the highlighting facilities of meta, an editor that allows jumping to the matching bracket, and manually reformatting go a long way towards understanding the output if you want to see all the details.

```
module output-syntax

imports
  tokens

exports
  sorts
    Output Line

  context-free syntax

  Line*          -> Output
```

Figure 7: v3/output-syntax file

### 4 Cycle 3: Semantic checking

After conversion of the input to the internal AST format, now it is time to check whether the input makes any sense. The goal of this cycle is therefore to perform semantic checking of the AST format.

Firstly, we need to define an output format that allows us to output our findings to the user. Secondly, we need to define functions that implement the semantic checking functionality, in particular:

1. Do all categories have an equal number of values?
2. Are all values uniquely defined (that is, in exactly one category in the entire problem)?
3. Do expressions contain only defined variables?

Note that due to the syntax restrictions we have defined in the input grammar, the program will not accept empty categories (that is, categories without a value are rejected as parse error). For this reason we do not need to include such checks here.

Not checking such restrictions does mean that you cannot always use the semantics checker in a different context, for example, to check the AST after loading it from another tool. There fore, in many cases it is a good strategy to include the check even if we know that it will never fail.

**Exercises:**

- Write a definition for the output format.
- Check that the number of values in each category is equal.
- Check that values are uniquely defined.
- Check that expressions only use defined variables.

#### 4.1 The output format

As you can see in Figure 7, the grammar of the output format is very small. It only states that there is output of the sort Output, and that the output consists of lines. The syntax for the sort Line will be defined in higher modules (technically, here the syntax for the Line sort is empty, and it will be extended in higher modules).

The first two checks all have to do with identifiers in categories. These will be dealt with in the next section. Then, we will deal with checks regarding use of variables in expressions.

```

equations

%%
%% hidden: check-semantic(Ast-problem) -> Output
%%
[cs-1] check-semantic($aprob)=check-counts(get-cats($aprob))
                                check-uniqness(get-cats($aprob))
                                check-defined($aprob)

%%
%% hidden: check-counts(Ast-category*) -> Line*
%%
[cc-1] check-counts($*acat1 $acat2 $*acat3 $acat4 $*acat5)=
                                check-counts($*acat1 $acat2 $*acat3 $*acat5)
                                when count-values(get-idents($acat2))=count-values(get-idents($acat4))

[cc-2] check-counts($*acat1 $acat2 $*acat3 $acat4 $*acat5)=
                                "Categories have different number of values"
                                when count-values(get-idents($acat2))!=count-values(get-idents($acat4))

[cc-3] check-counts($acat)=

%%
%% hidden: check-uniqness(Ast-category*) -> Line*
%%
[cu-1] check-uniqness($*acat1 $acat2 $*acat3 $acat4 $*acat5)=
                                "Identifier $iden2 used more than once"
                                when $*iden1 $iden2 $*iden3:=get-idents($acat2),
                                $*iden5 $iden2 $*iden6:=get-idents($acat4)

[deault-cu] check-uniqness($*acat)=

%%
%% hidden: count-values(Identifier*) -> Integer
%%
[cv-1] count-values()=0
[cv-2] count-values($iden $*iden)=1+count-values($*iden)

```

Figure 8: v3/check-semantic.asf file, part 1

```

"check-var-trav" ( Ast-expression*,Line*,Ast-category*)
    -> Line* {traversal(accu,continue,bottom-up)}

"check-var-trav" ( Ast-variable,   Line*,Ast-category*)
    -> Line* {traversal(accu,continue,bottom-up)}

```

Figure 9: Fragment of `v3/check-semantic.sdf` file

## 4.2 Checking values in categories

The `check-counts` function in Figure 8 repeatedly eliminates a category if it can find another category with the same number of identifiers. If it can find two categories with a different number of identifiers, it has found an error, and its output is a error message of the sort `Line`. If it has one category left, the input was correct, and the output of the function is empty.

Uniqueness checking can be expressed very compactly as shown in the `check-uniquess` function. What we need to find is an identifier that exists in one category, and then again in another. If this condition is matched an error is produced. If this condition cannot be met, the default equation is chosen (written as last rewrite rule of the function), and no output is generated, which means that everything is correct. As you can see, pattern matching can be very expressive. Writing the uniqueness checking code in an imperative language would be a lot more work. (Elsewhere we will see a less powerful side of functional programming.)

Also, I used ASF+SDF-generated access functions (the `get-X` functions) everywhere to obtain contents of sorts rather than a pattern match with the concrete syntax. For example the `get-iden` function of a category gives the list identifiers. The advantage of using these functions is that you get a layer of access functions on top of the concrete syntax of the `Ast-category` sort. When the syntax of the category ever changes, you can adapt the layer to using the new syntax rather than changing every occurrence of category syntax in your code.

## 4.3 Checking use of identifiers in expressions

The remaining semantic checks that have to be performed are checks in expressions, in particular, the question whether variables used in expressions use properly defined identifiers. The only way to check this is to walk through the expression trees, until you find a variable. At that point check whether the identifiers used in the variable are defined in different categories (we need ‘in different categories’ to exclude variables like ‘john/jeen’).

Programming the treewalk by hand means that for each node you may encounter in the tree, you should forward the walk down to its children, collect the output of the calls, and pass the result back up. In this example, writing a treewalk manually is still doable, since the number of nodes that you may encounter in a list of expressions is limited. However, ASF+SDF can generate these treewalks for you. The only two things you need to do are *a)* specify how you want to walk over the tree, and *b)* program the function explicitly for the nodes that need special treatment. The code is shown in Figures 9 and 10. In the `v3/check-semantic.sdf` file, two entries of the `check-var-trav` traversal function are defined (As convention, I use the `-trav` suffix to denote that this is a generated traversal function). The first entry is the starting point of the traversal. It takes a list of expressions, output collected so far (which is empty at the start as you can see in the `check-defined` wrapper function that initiates the treewalk), and the categories as a constant. The second entry is the special case. When we encounter a variable, we need to verify that the used identifiers are in two different categories. The special case is implemented in the equations. It starts with a check that we really have a variable node here. This is currently redundant, since the sort `Ast-variable` only contains syntax for variables, but when the sort is extended, this check ensures that this equation will not match any of the extensions. The second and third condition use the `find-iden` function

```

%%
%% hidden: check-defined(Ast-problem) -> Line*
%%
[cd-1] check-defined($aprob)=
        check-var-trav(get-exprs($aprob), ,get-cats($aprob))
%%
%% hidden: check-var-trav(Ast-variable,Line*,Ast-category*) -> Line*
%%                                     {traversal(accu,continue,bottom-up)}
%%
[cvt-1] check-var-trav($avar,$*line,$*acat)=$*line2 $*line3 $*line
        when is-var($avar)==true,
            <$*acat2,$*line2>:=find-iden(get-iden1($avar),$*acat),
            <$*acat3,$*line3>:=find-iden(get-iden2($avar),$*acat2)

%%
%% hidden: find-iden(Identifier, Ast-category*) -> <Ast-category*,Line*>
%%
[fi-1] find-iden($iden,$*acat1 $*acat2 $*acat3)=<$*acat1 $*acat3,>
        when $*iden1 $iden $*iden2:=get-idents($acat2)

[default-fi] find-iden($iden,$*acat)=<$*acat,"Identifier $iden undefined">

```

Figure 10: v3/check-semantics.asf file, part 2

in their match. This function takes an identifier, and a list of categories. If the identifier can be found in a category, it returns the remaining categories, and an empty list of error messages. If the identifier cannot be found, an error is produced and all categories are returned. By using the full list of categories in the second condition, and by using the result list of categories from the second condition in the third condition, we can be sure that a match will only be found in two different categories.

You can test the semantics checking code by writing an input term that calls the `check-semantics` function. Since the function uses AST format as its input, you either have to write AST format yourself, or use the `convert` function to transform it for you.

## 5 Cycle 4: Deducing variables

After parsing the input, transforming it to the AST format, and performing the static semantics checks, the time has come to start implementation of the reasoning mechanism.

The simplest form of reasoning is to deduce from an expression like ‘John/Paris’ that John has visited Paris (that is, deduce that the variable ‘John/Paris’ should be set to ‘true’), and from an expression like ‘not Jaap/tea’ that the ‘Jaap/tea’ should be set to ‘false’. The first thing to do is to create a data structure to store such information, that is, create a variable table for variables and their value. The second thing to do is to search the expression list of the problem for expressions like above, and use the information to fill the variable table. Therefore:

### Exercises:

- Make a table for storing variables and their values, and
- walk over a list of expressions to check whether we can extract information about the value of variables.

Since in the next iteration, we will want to perform this routine repeatedly, until no change occurs any more, we need to give an additional boolean output whether the walk changed anything.

```

module vartable-syntax

imports
  containers/Table[Ast-variable Ast-expression]
  ast-syntax-api

exports
  sorts
    Variable-table

  context-free syntax
    Table[[Ast-variable,Ast-expression]]          -> Variable-table

    "new-vtable"                                  -> Variable-table

    "get-var" (Variable-table,Ast-variable)       -> Ast-expression
    "has-var" (Variable-table,Ast-variable)       -> Boolean
    "add-var" (Variable-table,Ast-variable,Ast-expression) -> Variable-table

    "variables" ( Variable-table )                -> List[[Ast-variable]]

hiddens
  variables
    "$table"          [0-9]? -> Table[[Ast-variable,Ast-expression]]
    "$avar"           [0-9]? -> Ast-variable
    "$expr"           [0-9]? -> Ast-expression

```

Figure 11: v4/vartable-syntax.sdf file

### 5.1 Variable table

We can write our own table, but the library already supplies the basic functions, so we will use that instead. (It is also a good opportunity to use parameterized sorts in our application.) Figure 11 shows the syntax definition of the variable table. The real table is imported by the Table[Ast-variable Ast-expression] line. The sort that contains the table is Table[[Ast-variable, Ast-expression]] (note the double square brackets and the comma, compared to the import line).

Since I don't like the Table[[Ast-variable, Ast-expression]] sort, I encapsulated it in a new sort, namely Variable-table. Another way of getting rid of the Table sort (in the specification) would be to define an alias for the sort. However, you cannot nest aliases. Also, during execution, meta replaces the alias with its definition, which means that you will still see the original sort at run time. The disadvantage of introducing a new sort is that you have to re-implement all the functions.

Defining a new sort Variable-table has an additional advantage in this case. It allows me to extend the functionality of the standard table to deal with variables that have their identifiers swapped (that is, 'John/Paris' is treated the same as 'Paris/John'). Have a look in the equations file if you want to know how that was programmed.

### 5.2 Finding variable values

Although the word 'walk' in the introduction may have set you off in the direction of a treewalk, that is not appropriate here. The reason is that an expression list is only used at the top, as a component of a problem. Inside an expression, there are no lists of expressions. Additionally, even if there were, you would not be able to conclude anything from it.

The function deduce-vars that implements the deduction is shown in Figure 12. It calls deduce-var for each expression in the list. If the latter function returns that it has found a new

```

equations

%%
%% export: deduce-vars(Ast-expression*,Variable-table)
%%                                     -> <Ast-expression*,Variable-table,Boolean>
%%
[dvs-1] deduce-vars(,$vtable)=<,$vtable,false>

[dvs-2] deduce-vars($aexpr $*aexpr,$vtable)=<$*aexpr3,$vtable3,true>
  when <$vtable2,true>:=deduce-var($aexpr,$vtable),
  <$*aexpr3,$vtable3,$bool3>:=deduce-vars($*aexpr,$vtable2)

[dvs-3] deduce-vars($aexpr $*aexpr,$vtable)=
  <$aexpr $*aexpr3,$vtable3,$bool3>
  when <$vtable2,false>:=deduce-var($aexpr,$vtable),
  <$*aexpr3,$vtable3,$bool3>:=deduce-vars($*aexpr,$vtable2)

%%
%% hidden: deduce-var(Ast-expression,Variable-table)
%%                                     -> <Variable-table,Boolean>
%%
[dv-1] deduce-var($avar,$vtable)=<add-var($vtable,$avar,make-true()),true>
  when has-var($vtable,$avar)==false

[dv-2] deduce-var($aexpr,$vtable)=
  <add-var($vtable,$avar,make-false()),true>
  when is-not($aexpr)==true,
  $avar:=get-expr($aexpr),
  has-var($vtable,$avar)==false

[default-dv] deduce-var($aexpr,$vtable)=<$vtable,false>

```

Figure 12: v4/deduce-vars.asf file

```

equations

%%
%% export: simplify-exprs(Ast-expression*,Variable-table)
%%                                     -> <Ast-expression*,Boolean>
%%
[ses-1] simplify-exprs($*aexpr,$vtable)=<$*aexpr,false>
      when $*aexpr2:=simpl-expr-trav($*aexpr,$vtable),
            $*aexpr2==$*aexpr

[ses-2] simplify-exprs($*aexpr,$vtable)=<$*aexpr2,true>
      when $*aexpr2:=simpl-expr-trav($*aexpr,$vtable),
            $*aexpr2!=$*aexpr

%%
%% export: simpl-expr-trav(Ast-expression,Variable-table)
%%                                     -> Ast-expression {traversal(bottom-up,trafo,continue)}
%%
[set-1] simpl-expr-trav($avar,$vtable)=get-var($vtable,$avar)
      when has-var($vtable,$avar)==true

```

Figure 13: v5/simplify-expr.asf file, part 1

variable value in the expression (second value returned is true), the expression is eliminated from the list and the boolean value (third return value) returned becomes true to indicate that something has changed. If the `deduce-var` returns false as its second value, then nothing could be deduced from the expression, and it is kept in the list.

What you also see in the equations is my convention of using numbers in equation variables patterns, for example in `<$*aexpr3,$vtable3,$bool3>`. If a number of variables are assigned together in a condition, they all get the same number. In that way, it is easy to keep track of which variables belong together.

The `deduce-var` function itself is a list of cases, one for each expression form that can be used to compute the value of a variable. The default case covers all other expressions.

## 6 Cycle 5: Simplifying expressions

The next step is to use the knowledge obtained from the variable deduction by filling in the values. In addition, once variables have been replaced, it may be possible to simplify the expressions.

### Exercises:

- Perform a bottom-up treewalk to replace variables in the expressions.
- Extend the code to perform simplifications in the expressions.

### 6.1 Filling in variables

Essentially, this functionality is another treewalk. Since we modify the expressions, we need a transformation treewalk. Figure 13 shows the code. The code is very straightforward, except maybe for the detection of changes. As you can see, there is no modification boolean value carried around in the traversal function. Instead, we compare the list expressions before and after performing the traversal (by comparing `$*aexpr2` with `$*aexpr`). If they are equal, we conclude nothing has changed, otherwise, some rewriting has been performed during the walk.

```

%%
%% Expression simplifications
%%

%% and simplifications
[es-01] simpl-expr-trav(and(true(), $aexpr2), $vtable)=$aexpr2
[es-02] simpl-expr-trav(and(false(), $aexpr2), $vtable)=false()
[es-03] simpl-expr-trav(and($aexpr1, true() ), $vtable)=$aexpr1
[es-04] simpl-expr-trav(and($aexpr1, false()), $vtable)=false()

%% or simplifications
[es-11] simpl-expr-trav(or(true(), $aexpr2), $vtable)=true()
[es-12] simpl-expr-trav(or(false(), $aexpr2), $vtable)=$aexpr2
[es-13] simpl-expr-trav(or($aexpr1, true() ), $vtable)=true()
[es-14] simpl-expr-trav(or($aexpr1, false()), $vtable)=$aexpr1

%% not simplifications
[es-21] simpl-expr-trav(not(true()), $vtable)=false()
[es-22] simpl-expr-trav(not(false()), $vtable)=true()
[es-23] simpl-expr-trav(not(not($aexpr)), $vtable)=$aexpr

[es-24] simpl-expr-trav(not(or($aexpr1, $aexpr2)), $vtable)=
        and(not($aexpr1), not($aexpr2))
[es-25] simpl-expr-trav(not(and($aexpr1, $aexpr2)), $vtable)=
        or(not($aexpr1), not($aexpr2))

%% implies simplification
[es-31] simpl-expr-trav(implies($aexpr1, $aexpr2), $vtable)=
        or($aexpr2, not($aexpr1))

%% equal simplification
[es-41] simpl-expr-trav(equal($aexpr1, $aexpr2), $vtable)=
        or(and($aexpr1, $aexpr2), and(not($aexpr1), not($aexpr2)))

```

Figure 14: v5/simplify-expr.asf file, part 2

Although this may seem to be a complicated and expensive way to check whether rewriting occurred, it is in fact extremely cheap in meta due to a property of the underlying ATerm library.

## 6.2 Simplifications (part 1)

The simplifications are shown in Figure 14. The first thing that you may note is that I do not use the access functions `get-...()` and `is-...()`. I tried it, but I found it to be very unreadable compared to this version (it may be a nice exercise to try this yourself), so despite the disadvantage that syntactic changes in the AST expressions will break the code here, I threw the version with access functions away. The second thing you may notice is my numbering scheme of equations. Instead of numbering sequentially, I leave gaps between groups of equations. In that way, inserting another equation is much easier.

With respect to the simplification equations itself, the first eleven should be no surprise. The final four expressions are the interesting ones. The [se-24] and [se-25] equations push down the not over ‘and’ and ‘or’ operations, so they can be eliminated using the other ‘not’ simplifications once they are collected at the bottom. Since the push-down is performed in top-down fashion, and the traversal function runs bottom-up, pushing down is done one level at a time, which may be inefficient

```

%%
%% export: split-exprs(Ast-expression*) -> <Ast-expression*,Line*,Boolean>
%%
[spe-1] split-exprs($*aexpr1 $aexpr2 $*aexpr3)=<$*aexpr4,$*line4,true>
      when is-true($aexpr2)==true,
      <$*aexpr4,$*line4,$bool4>:=split-exprs($*aexpr1 $*aexpr3)

[spe-2] split-exprs($*aexpr1 $aexpr2 $*aexpr3)=
      <$*aexpr4,"False expression detected" $*line4,true>
      when is-false($aexpr2)==true,
      <$*aexpr4,$*line4,$bool4>:=split-exprs($*aexpr1 $*aexpr3)

[spe-3] split-exprs($*aexpr1 $aexpr2 $*aexpr3)=<$*aexpr4,$*line4,true>
      when is-and($aexpr2)==true,
      <$*aexpr4,$*line4,$bool4>:=split-exprs($*aexpr1 get-lft($aexpr2)
      get-rgt($aexpr2) $*aexpr3)

[default-spe] split-exprs($*aexpr)=<$*aexpr,,false>

```

Figure 15: v5/simplify-expr.asf file, part 3

in some situations.

The final two simplifications are not really simplifications but eliminations. The equations eliminate the ‘implies’ and ‘equal’ operators. Therefore, it would be enough to run them once only instead of repeatedly (as we will do further in the development). For large number of expressions this difference may be noticeable (I have not tried this). However, since the program is not aimed at solving large puzzles efficiently, I found it easier to put them here. Also, having them here has the advantage that I can use the ‘implies’ and the ‘equal’ operators in expressions whenever I feel like it, these two equations will make sure they will be eliminated at the first simplification traversal. (In the alternative solution, the elimination is a separate step that has to be called separately at the appropriate point in the program.)

### 6.3 Simplifications (part 2)

There are a number of other simplifications that should be performed in the expression list. The expression ‘true’ will not provide any useful information, and can be eliminated. The expression ‘false’ is almost the same, except that this should never happen, so this should result in an message to the user. Finally, an ‘and’ operation at the top of an expression can be eliminated by considering the left and right sides separately (if ‘ $a$  and  $b$ ’ must evaluate to true, then  $a$  as well as  $b$  must evaluate to true on their own). The code for these simplifications is shown in Figure 15.

## 7 Cycle 6: Gluing everything together

In the past iterations, we have constructed functions that solved a part of the puzzle. As a reminder, below is the list of functions that we have constructed.

- `convert(Problem)`  $\rightarrow$  `Ast-problem` converts a logic-puzzle problem to AST format in the `convert` module.
- `check-semantic(Ast-problem)`  $\rightarrow$  Output checks whether the static semantic requirements are met, and reports errors to the output in the `check-semantic` module.

- `deduce-vars(Ast-expression*, Variable-table) → ⟨Ast-expression*, Variable-table, Boolean⟩` deduces the value of variables from simple expressions in the `deduce-vars` module.
- `simplify-exprs(Ast-expression*, Variable-table) → ⟨Ast-expression*, Boolean⟩` fills in the variables, and simplifies or eliminates expressions in the `simplify-expr` module.
- `split-exprs(Ast-expression*) → ⟨Ast-expression*, Line*, Boolean⟩` splits expressions, so they can be considered individually, also in the `simplify-expr` module.

The next objective is to glue all these functions together into one main function `solve-problem`.

### Exercises:

- Write the `solve-problem` function.

## 7.1 Writing the `solve-problem` function

The steps that the main function should take are

1. Convert the input to AST format
2. Check the correctness of the AST data. If errors are reported, end the function and return the errors to the user.
3. Try to solve the given problem by reducing the expressions. This means do repeatedly
  - (a) deduce new values from the expressions,
  - (b) split the expressions, and
  - (c) simplify the expressions

until we either have no expressions left, or until we make no progress any more.

4. Output the results to the user.

The code that implements this functionality is shown in Figure 16. The function performs the first two steps, and depending on the output, it either returns the output, or it hands the execution over to the `solve-exprs` function. This function performs expression reduction once. It then hands over the results to the `solve-exprs2` function. The latter decides whether the end has been reached or not. If not, the reduction is performed again.

This solution is quite nice, however it also shows one of the weaker points of functional programming. In an imperative language one would use a while loop in a single function, instead of the functions defined here.

If you test the program with an input file like (file `v6/puzzle.trm`)

```

solve-problem(
  { jaap, john, jacob, jeen }
  { paris, london, brussel, berlin }
  { coffee, tea, beer, wine }

  not jaap/tea
  not jaap/coffee
  jeen/wine
)

```

```

equations

%%
%% export: solve-problem( Problem ) -> Output
%%
[sp-1] solve-problem($prob)=$output
      when $aprob:=convert($prob),
            $output:=check-semantics($aprob),
            $output!=

[sp-2] solve-problem($prob)=solve-exprs($aprob,new-vtable)
      when $aprob:=convert($prob),
            $output:=check-semantics($aprob),
            $output==

%%
%% hidden: solve-exprs( Ast-problem, Variable-table ) -> Output
%%
[se-1] solve-exprs($aprob,$vtable)=
        solve-exprs2($aprob,$vtable2,$*aexpr4,$*line3,
                    $bool12|$bool13|$bool14)

      when $*aexpr:=get-exprs($aprob),
            <$*aexpr2,$vtable2,$bool12>:=deduce-vars($*aexpr,$vtable),
            <$*aexpr3,$*line3,$bool13>:=split-exprs($*aexpr2),
            <$*aexpr4,$bool14>:=simplify-exprs($*aexpr3,$vtable2)

%%
%% hidden: solve-exprs2(Ast-problem,Variable-table,
%%                      Ast-expression*,Output,Boolean) -> Output
%%
[se2-1] solve-exprs2($aprob,$vtable,$*aexpr,$output,$bool)=$output
      when $output!=

[se2-2] solve-exprs2($aprob,$vtable,$*aexpr,,false)=
        "Variables found: $vtable "

[se2-3] solve-exprs2($aprob,$vtable,,,true)=    "Variables found: $vtable "

[se2-4] solve-exprs2($aprob,$vtable,$*aexpr,,true)=
        solve-exprs(set-exprs($aprob,$*aexpr),$vtable)

```

Figure 16: v6/solve-problem.asf file

you get output like "Variables found: [`<var(jeen, wine), true()>`, `<var(jaap, coffee), false()>`, `<var(jaap, tea), false()>`]". However, the program is not reasoning further. For example, since Jeen drinks wine, why does the program not conclude that Jaap, John, and Jacob do not drink wine? The answer is that we have not told it that it may draw this conclusion. What is missing is the restriction that exactly one person drinks wine, or in formula:

```
(jaap/wine and not(john/wine or jacob/wine or jeen/wine ))
or (john/wine and not(jaap/wine or jacob/wine or jeen/wine ))
or (jacob/wine and not(jaap/wine or john/wine or jeen/wine ))
or (jeen/wine and not(jaap/wine or john/wine or jacob/wine))
```

If you add this to the input term (done in file `puzzle2.trm`), you will find that the program does draw the conclusion that all other persons do not drink wine. A similar story can be told about drawing conclusions between three different variables, for example, given that ‘Jaap/beer’ and ‘beer/Berlin’ hold, the program should conclude that ‘Jaap/Berlin’ is also true.

Besides the lack of expressions needed for reasoning, the program also still lacks decent output; dumping a list of variables in AST format onto the screen is not very useful for the average user. These two extensions are implemented in the following two iterations.

## 8 Cycle 7: Reasoning expressions

At the end of the previous cycle, we established the need for having expressions that allows the program to reason. In principle, the user can add them by hand, however it is a lot of boring work with a high risk of making mistakes. To me, that sounds like a perfect job for the computer. On the other hand, there may be occasions that a user does not want to have the expressions added by the program. The solution to this dilemma is to give the user a choice by adding two keywords to the input language, namely ‘basic-relations’ and ‘one-step-deductions’. The former causes the program to add the basic relations like ‘there is exactly one person that drinks wine’, the latter adds relations like ‘if person *A* is in city *B*, and the person in city *B* has drink *C*, then person *A* has drink *C*’. If the user does not enter the new keywords, no reasoning expressions are generated.

These two new keywords behave like an entire expression, that is, one should be able to write something like

```
solve-problem(
  { jaap, john, jacob, jeen }
  { paris, london, brussel, berlin }
  { coffee, tea, beer, wine }

  basic-relations
  one-step-deductions

  not jaap/tea
  not jaap/coffee

  jeen/wine

  not tea/london
  not tea/paris

  beer/berlin

  john/paris
)
```

Category+ Relation*	-> Problem
Expression	-> Relation
"basic-relations"	-> Relation {constructor}
"one-step-deductions"	-> Relation {constructor}

Figure 17: Major changes in the `logi-syntax` module.

"expr" "(" expr: Ast-expression ")"	-> Ast-relation {cons("expr")}
"basic-relations" "(" ")"	-> Ast-relation {cons("basic")}
"one-step-deductions" "(" ")"	-> Ast-relation {cons("deduct")}

Figure 18: Major changes in the `ast-syntax` module.

However, they are not of sort `Expression`, because the user would be allowed to use the new keyword anywhere in an expression. Instead, we need a new sort between the `Problem` and the `Expression` sorts in the input language. I call this sort `Relation`. A `Problem` now has categories and relations. A relation is either an expression, or it is one of the new keywords. During the simplification loop, the new keywords can be expanded to a list of expressions.

#### Exercises:

- Introduce the relation sort into the program by refactoring (that is, introduce the `Relation` sort into the entire program by changing code without changing the functionality).
- Add generators to the program that replace the new keywords by their lists of expressions.

## 8.1 Introducing the `Relation` sort

Here, I will not show all the details. There are just too many, mainly non-interesting, changes. If you really want to know, run a `diff` command between the `v6` and the `v7` directories.<sup>1</sup> Instead, I will give you a short list, and go into more detail with the more interesting changes.

- `logi-syntax` module: As shown in Figure 17, the biggest change is that a `Problem` is now a combination of categories and relations. A relation is an expression or one of the new keywords.
- `ast-syntax` module: The same change also happens in the AST syntax. Figure 18 shows only the new production rules for the `Ast-relation` sort.
- The `ast-syntax-api` module should be re-generated to make the new `Ast-relation` sort accessible through access functions.
- The `convert` module is also adapted (`convert-exprs` becomes `convert-rels`).
- The traversal function in `check-semantics` module now walks over relations instead of expressions.
- The `split-exprs` function in the `simplify-expr` module now splits relations rather than expressions. For example, the equation that splits an expression with an ‘and’ as top-level operand changes to the code shown in Figure 19. As you can see, compared to the original code in Figure 15, the equation matches on a list of (AST-)relations instead of expressions. The

<sup>1</sup>Using GNU `diff`, something like `diff -r v6 v7` should work.

```
[spe-3] split-exprs($*arel1 $arel2 $*arel3,$*acat)=<$*arel4,$*line4,true>
  when is-expr($arel2)==true,
    $aexpr2:=get-expr($arel2),
    is-and($aexpr2)==true,
    <$*arel4,$*line4,$bool4>:=split-exprs($*arel1
                                          make-expr(get-lft($aexpr2))
                                          make-expr(get-rgt($aexpr2))
                                          $*arel3,$*acat)
```

Figure 19: Modified equation in the `split-exprs` function.

```
[spe-4] split-exprs($*arel1 $arel2 $*arel3,$*acat)=<$*arel5,$*line5,true>
  when is-basic($arel2)==true,
    $*arel4:=gen-basics($*acat,),
    <$*arel5,$*line5,$bool5>:=
      split-exprs($*arel1 $*arel3 $*arel4,$*acat)

[spe-5] split-exprs($*arel1 $arel2 $*arel3,$*acat)=<$*arel5,$*line5,true>
  when is-deduct($arel2)==true,
    $*arel4:=gen-onesteps($*acat,,),
    <$*arel5,$*line5,$bool5>:=
      split-exprs($*arel1 $*arel3 $*arel4,$*acat)
```

Figure 20: Adding calls to the keyword generators.

first condition checks that it is an expression, and the second condition extracts the expression from the relation. Then the old code is used, except for some changes from expression variables to relation variables.

- The other simplification functions also change from expression to relation in much the same way.
- Finally, the `solve-problem` module passes relations down to the simplification functions.

## 8.2 Generating reasoning expressions

The next goal is to add generators to the program that replace the new keywords with their equivalent reasoning expressions. The first thing to note is that the generators need the categories to do their work. Luckily, this information is available in the `Ast-problem` sort near the root of the AST tree. The second thing to note is that keyword replacements can be considered to be a form of simplification, and nicely fit in the expression simplification process we have been programming (although it may be more appropriate to call it ‘relation simplification process’ since our refactoring).

Because the keywords only occur at the upper-most level, the best place for the new generators seems to be in the `split-exprs` function. If you paid close attention to the previous refactoring step, you may have noticed that I paved the way for the new generators by adding the categories as a second parameter of the function. Adding calls to the generators is straightforward now, as can be seen in Figure 20. When detecting a keyword, a call is made to the appropriate generator, and the keyword is replaced by the output of the generator.

The generators themselves, especially the `gen-basics` generator, are a different story. Rather than explain them in full length, I will only describe them briefly. Together with the code, you should be able to work out how it is done up to the level that you want to know.

```

%%
%% hidden: gen-basics(Ast-category*,Ast-category*) -> Ast-relation*
%%
%% Call: gen-basics(<categories>,)
%%
[fec-1] gen-basics(,$*acat)=
[fec-2] gen-basics($acat1 $*acat1, $*acat2)=
        gen-oneof($acat1,acats-idents($*acat1 $*acat2))
        gen-basics($*acat1,$acat1 $*acat2)

%%
%% hidden: gen-oneof(Ast-category, Identifier*) -> Ast-relation*
%%
[gof-1] gen-oneof($acat,)=
[gof-2] gen-oneof($acat,$iden $*iden)=
        gen-basic-relation(get-idents($acat),$iden)
        gen-oneof($acat,$*iden)

%%
%% hidden: gen-basic-relation(Identifier*,Identifier) -> Ast-relation*
%%
[gbr-1] gen-basic-relation(,$iden3)=
[gbr-2] gen-basic-relation($iden1,$iden3)=make-expr(make-var($iden1,$iden3))
[gbr-3] gen-basic-relation($iden1 $+iden2,$iden3)=
        make-expr(gen-basic-expression($iden1 $+iden2,,$iden3))

%%
%% hidden: gen-basic-expression(Identifier+,Identifier*,Identifier)
%%                                     -> Ast-expression
%%
[gbe-1] gen-basic-expression($iden1      , $+iden3,$iden4)=
        make-and(make-var($iden1,$iden4),
        make-not(make-ors(
                make-variables($+iden3,$iden4))))
[gbe-2] gen-basic-expression($iden1 $+iden2,$*iden3,$iden4)=
        make-or(make-and(make-var($iden1,$iden4),
        make-not(make-ors(
                make-variables($+iden2 $*iden3,$iden4))))),
        gen-basic-expression($+iden2,$iden1 $*iden3,$iden4))

```

Figure 21: The `gen-basics` generator function.

The basic relations generator is in module `simplify-expr`, and shown in Figure 21. Not shown in the figure are a number of support functions. The first one is the `acats-idens` function, which flattens the given categories to a list of identifiers. The second support function is the `make-variables` function. It constructs a list of expressions from pairs of identifiers, for example `'make-variables(X Y, Q)'` generates the list `'X/Q Y/Q'`. The third and last support function is `make-ors`. It takes a list of expressions, and combines them to a single expression with 'or' operands in between.

The basic approach of the generator is to take each category in turn, and for each category and each identifier in all other categories, generate an expression like the one in Section 7.1 (the expression shown there is for the category 'Persons' and the identifier 'wine'). This is done in equation [gbr-3] of the `gen-basic-relation` function. (The category is the first parameter, only its identifiers are transferred instead because further down we need to access the identifiers individually.) The other equations in the function are for pathological cases like one category or one identifier in a category.

An expression like the one in Section 7.1 is a sequence of 'or' operations at the top (generated by the outermost `make-or` application in equation [gbe-2]). A single sub-expression (constructed in [gbe-1] as well as in [gbe-2]) consists of an `$iden1/$iden4 and not(...)` expression, with the dots replaced by another 'or' sequence of variables containing the other identifiers in the category (in the `iden2` and `iden3` variables), and identifier `$iden4`. Construction of this latter 'or' sequence is done using the `make-variables` and `make-ors` support functions.

The one-step deductions generator is also in module `simplify-expr`, and shown in Figure 22. The basic operation of the generator is to take each category, take each category from the remaining categories, and call the `gen-implies-exprs1` function. This function then generates a list of expressions of the form `'a/b => a/c = b/c'` (for example for `a=Jeen`, `b=wine`, `c=Paris`, the expression would be (in words) 'if Jeen/wine, then Jeen/Paris should be equal to wine/Paris'). Such an expression is generated for all combinations of `a`, `b`, and `c`, where `a` is an identifier from the first selected category, `b` an identifier from the second selected category, and `c` an identifier from the remaining categories. Since loops have to be implemented by recursive function calls, three functions are needed.

## 9 Cycle 8: Output format

The variable-dump that the program outputs as result is not very readable, instead we want something more compact. In particular, we are only interested in the positive relations that the program can find, ordered such that related values from different categories are shown together. For example, output like `{london, wine, jeen} {berlin, jaap, beer} {brussel, jacob, tea} {coffee, john, paris}`.

### Exercises:

- Generate output like above from the variable information in the variable table.

### 9.1 Making a compact output format

The standard approach of writing a syntax definition, followed by implementing a conversion function also applies here. However, instead of writing it all ourselves, I used the library to define the syntax, in particular the templated sort `Set`. The relevant syntax definitions are shown in Figure 23. By injecting the set of identifiers into the `Line` sort, we add the set to the output syntax. The output shown above is a list of such sets, and thus a list of lines.

The rewrite rules are shown in Figure 24. There are three steps to compute and convert the output. The first step is to extract all the 'true' variables from the variable table, and insert them into the list of identifier sets. The second step, implemented in the `simplify-list` function, searches the list of sets, and combines two sets when they have a common element. Finally, the `convert-to-lines` function convert the list of sets to a list of lines, which is compatible with the `Output` sort.

You may wonder why I bother with the conversion function, why not use the following:

```

%%
%% hidden: gen-onesteps(Ast-category*,Ast-category*,Ast-category*)
%%                                     -> Ast-relation*
%% call: gen-onesteps(<categories>, ,)
%%
[gdd-1] gen-onesteps(,$*acat3,$*acat5)=
[gdd-2] gen-onesteps($acat1 $*acat2,,$*acat5)=
        gen-onesteps($*acat2,$acat1 $*acat5,)
[gdd-3] gen-onesteps($acat1 $*acat2,$acat3 $*acat4,$*acat5)=
        gen-implies-exprs1(get-idents($acat1),get-idents($acat3),
                acats-idents($*acat2 $*acat4 $*acat5))
        gen-onesteps($acat1 $*acat2,$*acat4,$acat3 $*acat5)

%%
%% hidden: gen-implies-exprs1(Identifier*,Identifier*,Identifier*)
%%                                     -> Ast-relation*
%%
%%
[gi1-1] gen-implies-exprs1(,$*iden3,$*iden4)=
[gi1-2] gen-implies-exprs1($iden1 $*iden2,$*iden3,$*iden4)=
        gen-implies-exprs2($iden1,$*iden3,$*iden4)
        gen-implies-exprs1($*iden2,$*iden3,$*iden4)

%%
%% hidden: gen-implies-exprs2(Identifier,Identifier*,Identifier*)
%%                                     -> Ast-relation*
%%
%%
[gi2-1] gen-implies-exprs2($iden1,,$*iden4)=
[gi2-2] gen-implies-exprs2($iden1,$iden2 $*iden3,$*iden4)=
        gen-implies-exprs3($iden1,$iden2,$*iden4)
        gen-implies-exprs2($iden1,$*iden3,$*iden4)

%%
%% hidden: gen-implies-exprs3(Identifier,Identifier,Identifier*)
%%                                     -> Ast-relation*
%%
%%
[gi3-1] gen-implies-exprs3($iden1,$iden2,)=
[gi3-2] gen-implies-exprs3($iden1,$iden2,$iden3 $*iden4)=
        make-expr(make-impl(make-var($iden1,$iden2),
                make-eql(make-var($iden1,$iden3),
                        make-var($iden2,$iden3))))
        gen-implies-exprs3($iden1,$iden2,$*iden4)

```

Figure 22: The gen-onesteps generator function.

```

imports
  variable-syntax
  output-syntax
  containers/Set[Identifier]

exports
  context-free syntax
    "generate-list" ( Variable-table )          -> Output

    Set[[Identifier]]                            -> Line

```

Figure 23: Part of the syntax definitions of the `gen-list` module.

```

equations

%%
%% export: generate-list(Variable-table) -> Output
%%
[gl-1] generate-list($vtable)=convert-to-lines(simplify-list(
    insert-vars(,variables($vtable),$vtable)))

%%
%% hidden: insert-vars(Set[[Identifier]]*,List[[Ast-variable]],
%%                    Variable-table)          -> Set[[Identifier]]*
%%
[iv-1] insert-vars($*idset,[$avarlst1,$avar2,$avarlst3],$vtable)=
    insert-vars($*idset {get-iden1($avar2),get-iden2($avar2)},
    [$avarlst1,$avarlst3],$vtable)
    when is-true(get-var($vtable,$avar2))==true

[default-iv] insert-vars($*idset,$values,$vtable)=$*idset

%%
%% hidden: simplify-list(Set[[Identifier]]*) -> Set[[Identifier]]*
%%
[sl-1] simplify-list($*idset1 $idset2 $*idset3 $idset4 $*idset5)=
    simplify-list($*idset1 $idset9 $*idset3 $*idset5)
    when {$idenlst1,$iden2,$idenlst3}:=$idset2,
    {$idenlst5,$iden2,$idenlst7}:=$idset4,
    $idset9:={$idenlst1,$iden2,$idenlst3,$idenlst5,$idenlst7}

[default-sl] simplify-list($*idset)=$*idset

%%
%% hidden: convert-to-lines(Set[[Identifier]]*) -> Line*
%%
[ctl-1] convert-to-lines()=
[ctl-2] convert-to-lines($idset $*idset)=$idset convert-to-lines($*idset)

```

Figure 24: `v8/gen-list.asf` file.

<code>Set[[Identifier]]*</code>	<code>-&gt; Output</code>
---------------------------------	---------------------------

In this way, the output of `simplify-list` is immediately compatible without conversion. While this is true, it has the disadvantage that the empty output can now be written in two ways (as zero lines, or as zero `Set[[Identifier]]s`), and has become ambiguous. That means that a condition like `$output==` cannot be used any more. Although it can be done in this way, it is easier to prevent the ambiguity from happening. I therefore chose to make a single set a single line, and have a conversion function that converts a list of sets to a list of lines.

Hooking the `generate-list` function into the program is done by replacing the generation of a line with the variable table (in equations [se2-1] and [se2-2] in the `solve-problem` module) by a call to the function. (Do not forget to delete the syntax of the deleted line.)

## 10 Next steps

A program is never complete, and `logisolve` is no exception.

One of the things you can do is to do more refactoring. For example, the combination of the problem and the variable table seem to belong together, yet they are two separate entities in the program.

Expansion of the program can for example be done by allowing more complex relations. A big step forwards would be to allow expressing relations like ‘Mary traveled longer than Elizabeth’, or ‘the ticket for the trip to the zoo was two euro more expensive than the bill for diner’. You could also try to cover the authoring-side of these puzzles, that is, expand the program such that it can be used to design these puzzles. Your first goal there may be clue-checking, which answers questions like ‘Can we drop some of the relations and still solve the puzzle?’.

Another direction is to try speeding up the program. The almost trivial step here is to try to compile the application (which is a good exercise in itself). More difficult speedups can be found in improving the used simplification algorithm. In the current implementation, we try rewriting all expressions each time. By adding additional information (for example, keep a list of used variables with each expression), you can skip rewrite attempts that are useless.

Last but not least, you could also try to attach a new back-end to the program and generate the nice graphical matrices that normally come with these puzzles.

## 11 History

You may ask, did I really solve the logic-puzzle program in the way described above? Well, almost. Below, I will tell you some of the history.

I did implement everything (except the pretty-output function) for the first time in a few days without writing any documentation, saving the program source after each cycle in a separate directory to allow me to work out afterwards what I did in each cycle. Afterwards I reconstructed the order to be:

After defining the input syntax (also Cycle 1 here), I first implemented semantics checking (Cycle 3 here) using the input format. I did include the ‘basic-relations’ and ‘one-step-deductions’ primitives immediately (here introduced in Cycle 7). Then I realized I needed an AST format, so I took the boolean expressions from the library as data format, and implemented the conversion (Cycle 2 here). During the conversion, I eliminated the ‘implies’ and the ‘equal’ operators (now part of Cycle 5), because I thought I would not need them again. After implementing the conversion I thought about refactoring the checking code to use the AST format but decided against it. Instead I introduced the variables just as in this document (Cycle 4 here). I knew that I needed the reasoning expressions, so before building the other simplifications, I implemented the functions that generate the reasoning expressions (Cycle 7 here) for the first time. Then I did the simplifications (Cycle 5) and the loop

(Cycle 6). After some experiments, I found that my reasoning expressions were not working so I fixed that by almost completely re-implementing them. I never got around to implementing the pretty-output function the first time.

During this first attempt I discovered that performing semantics checking in the input format was not a good idea, I should have done that using the AST format (like I describe in this document). The second thing I discovered was during implementation of the reasoning expression generator functions. I discovered that I needed the ‘implies’ and ‘equal’ operators again, but I had no rewrite function available for them at that point of the conversion to the AST format. As a result, in this version I have both operators available in the AST, along with their simplifications. The third thing I discovered was that using the boolean expressions provided by meta as base for expressions in the AST was a bad choice. I expected to get a considerable advantage because that module already contained a number of simplification equations. However, the advantage was less than I expected. In the end the AST format for expressions was a big mess. As a result, in this version I defined my own expressions.

Rather than using the original order in this document and then having to explain all the messy changes that are really confusing at first sight, I decided to implement everything for the second time in parallel with writing this document (although I should confess that I have a number of scripts that allow me to go back in time and fix problems in earlier cycles without much effort). I incorporated the lessons I learned the first time to make it a less confusing document and increase the chance that you understand what I am doing.

That does not mean that the current implementation is ideal. If I were to re-implement the program again, I would probably do a number of things slightly different. For example, maybe a ‘xor’ operator would be a better choice to express the reasoning expressions instead of the large expression at the end of Cycle 6. Also, I am not sure that I made the right choice for the refactoring I explain in Cycle 7 (where I add the new ‘basic-relations’ and ‘one-step-deductions’ primitives). Maybe they should have been added from the beginning instead like I did in the first attempt.

## References

- [vdBK03] M.G.J. van den Brand and P. Klint. *ASF+SDF Meta-Environment User Manual*. Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, September 2003. Revision 1.134.