Content Planner Module for Talking Robots

Bernd Kiefer, DFKI Saarbrücken

Version 1.2, January 17, 2018

Contents

1 Installation **2**

2 Execution **3**

3 Project File **4**

4 GUI Mode **6**
4.1 Main Window 6
4.2 Trace Window 8
4.3 Batch Test Window 9

5 General Description **11**

6 Rules **12**
6.1 Matching Part 12
6.2 Actions 14
6.3 Some Simple Examples 16

7 Build-in Functions **17**
7.1 Mathematical Functions 17
7.2 Other Functions 17

8 Adding New Functions **18**

9 Processing Regime **19**

A More Examples **20**
A.0.1 Adding to relations the wrong way 23
A.0.2 Global variable “maps” 24
A.0.3 All you can do with variables 24
A.0.4 Use of functions for tests and results 25

Chapter 1

Installation

You obviously already have cloned the project from github. To build it, you have to install Java 8 and Maven on your machine. Then, in the best case, a simple

```
mvn install
```

should fetch all dependencies needed and build the project. If you experience problems downloading the dependencies, you may have to clone and build some other projects manually first:

```
git clone https://github.com/bkiefer/dataviz.git
git clone https://github.com/bkiefer/openccg.git
git clone https://github.com/bkiefer/j2emacs.git
```

To use the Emacs features, Mac users have to use AquaMacs. Plain X11 Emacs does not seem to provide a server mode. Windows Emacs should work, but has not been tested lately.

Chapter 2

Execution

The utterance planner is started with the following command:

```
cplanner [-[g]<enerate batch> batchfile]
         [-[G]<enerate all sentences> batchfile]
         [-[a]<nalyze batch> inputfile]
         [-[p]<lan batch> inputfile]
         [-[P]<lan all batch> inputfile]
         [-i<nteractive shell>] [-t<race>={1,2,3}] [-e<macs>]
         <projectfile> [batchoutput]
```

Simply typing `cplanner` will open an empty main window and an Emacs, if installed.
`projectfile` is a file that contains settings, like the path to a CCG grammar, and a list of all rule files that should be loaded. The structure of the project file is described in more detail in section 3.
By default, the planner reads the project file, and the rule files specified there, and continues in an interactive console mode with rudimentary readline support. The switches `-g`, `-G`, `-p`, and `-P` let the planner run in batch modes, where the uppercase versions try to come up with all possible variants when the `random` function is used in the rules. The difference between `-g` and `-p` is that `-g` attempts to do text generation (with a CCG grammar, if available), while `-p` stops after the graph rewriting process and outputs the result graph.
The `-t` flag controls tracing onto the console, using a bitmask for different phases, bit 1 to trace the match phase and bit 2 to trace the application phase. Thus, `-t 3` will show trace information for the matching as well as the applicaton phase. The `-t` flag can be given together with `-g`. In this case, tracing to the console will be used only when the *process* button is used, and not when the GUI tracing mode itself is activated.
The `-i` and `-e` flags are the default mode when running without giving any arguments and will open the GUI and connect it to a newly started Emacs client that will show the result of rule loading, making files and errors mouse sensitive to facilitate the editing and debugging of rule files. In Unix/Linux there is usually no need to specify the path to the Emacs binary. This is different on Macs, because only AquaMacs works as emacs server, the path to the AquaMacs binary must be specified. If the Emacs connection is properly set up, the GUI trace window described below allows to jump directly to the definition of rules used during processing.

Chapter 3

Project File

The project file is a plain text file that is divided into different sections. The sections are separated by lines specifying the section names, which are enclosed in square brackets, i.e., `[Settings]` for the section containing the project settings. Empty lines are skipped, and everything following a hash character (‘`#`’) until the end of line is treated as comment and ignored.

```
# relative paths are relative to the location of this file

[Settings]
history_file = history.txt
encoding = ISO-8859-15
ccg_grammar = ../../resources/grammars/openccg/grammar.xml

[Rules Stage 1]
attributes.trf
discoursemarkers.trf
infostruct.trf
transitivity.trf

[Rules Stage 2]
callfun.trf
test.trf
foo.bar
```

Figure 3.1: Sample Project File

Currently, the content planner distinguishes two kinds of sections, the `Settings` section and the `Rules` section, of which there may be more than one.
The `Settings` section contains key-value pairs that are separated by the equal sign (‘`=`’). Currently, valid settings are the path to the corresponding CCG grammar, whose key must be `ccg_grammar`, a `history_file` to store recent inputs, and an `encoding` for the grammar files. If no encoding is specified, `UTF-8` is taken as a default.
The section names for rules have to start with the word `Rules`, the rest before the closing bracket is taken as the stage name for the rule list that is following the section separator.

In a rule section, every non-empty line must contain the path to one rule file. Paths can be given either as absolute paths, or relative to the location where the project file is located. For a description in what order the rules are processed and what can be done with different stages, see section 9.

Chapter 4

GUI Mode

Figure 4.1 shows all windows opened by the GUI in one picture. The topmost window is the GUI main window, which is disabled because the trace window below it is present and acts as a modal window for this processor. In the bottom left corner is the Emacs window that is connected to the GUI, with the rule loading output in the lower half and a rule file for editing in the upper half.
When started in GUI mode, the application looks for the file `.cplanner` in the application directory. This file has the same syntax as the project file described above. It also has a `Settings` section to set two of the the command line arguments permanently, so that they do not have to be provided on every startup, and a `RecentFiles` section, which is maintained by the application, that contains the recently opened project files.
Figure 4.2 shows a sample `.cplanner` file.

4.1 Main Window

The main window, shown in figure 4.3, consist of the input area at the top, and the input and output LF displays below it, and a status bar for status and error messages at the bottom. The function of the buttons in the tool bar and the menu items will be described in what follows.

File Menu

When running in GUI Mode, it is not necessary (put possible) to specify the project file on the command line. The file can also be opened using the `File > Open` menu item. `New` will open a completely fresh planner with empty rule set, which means that a rule file must be opened from the new window for the planner to work. The `File > Recent Files` submenu will contain the last project files that have been opened.

Button Toolbar

The button toolbar is located below the menu bar, and contains buttons for, in this sequence:

- Reloading the rule set (if it has been changed in an editor)
- Processing the logical form in the input area

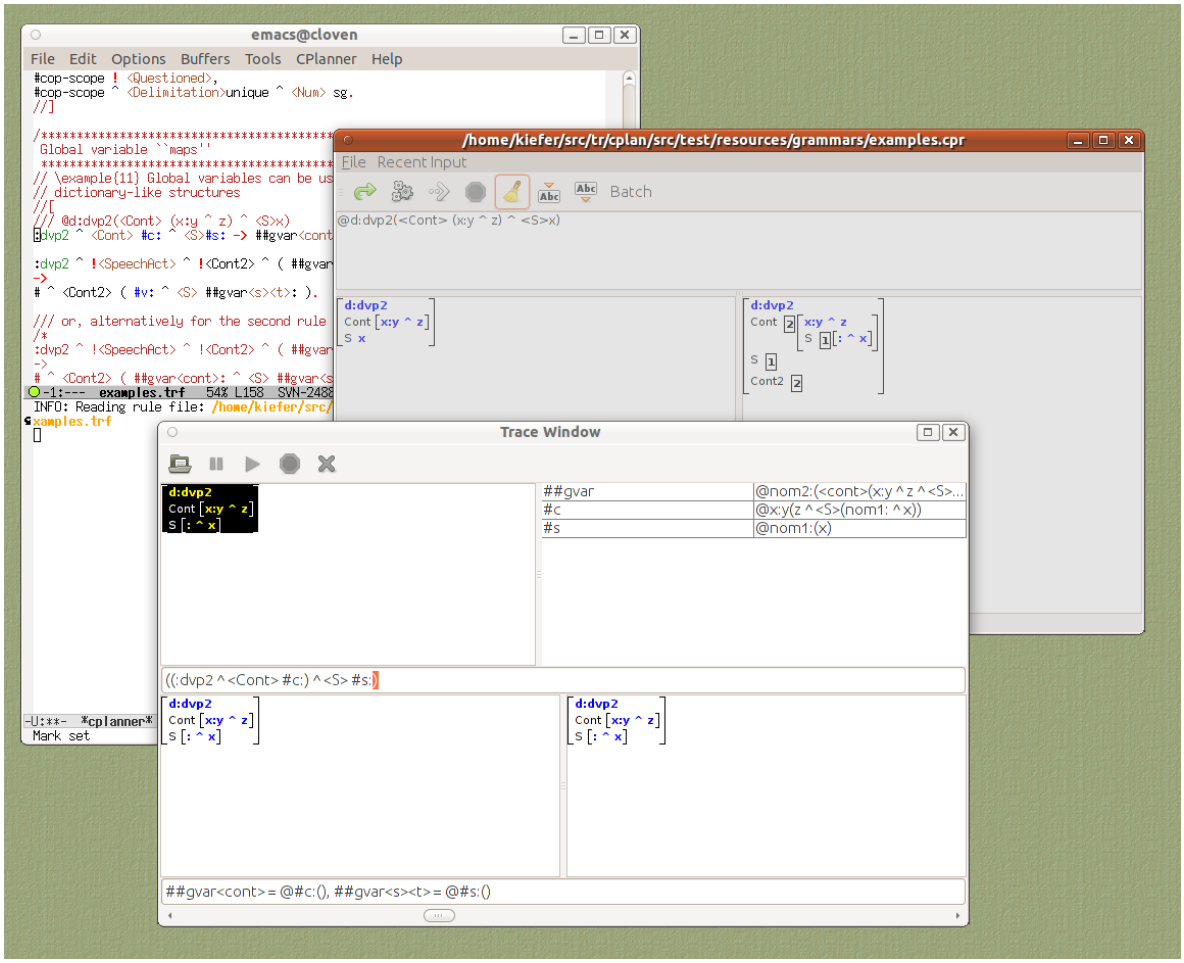

Figure 4.1: Content Planner Windows

```
# relative paths are relative to the location of this file

[Settings]
tracing = none # none / match / action / all
emacs = /home/kiefer/Applications/AquaMacs/bin/Aquamacs

[RecentFiles]
Rules/allrules.cpj
```

Figure 4.2: Sample Preferences File

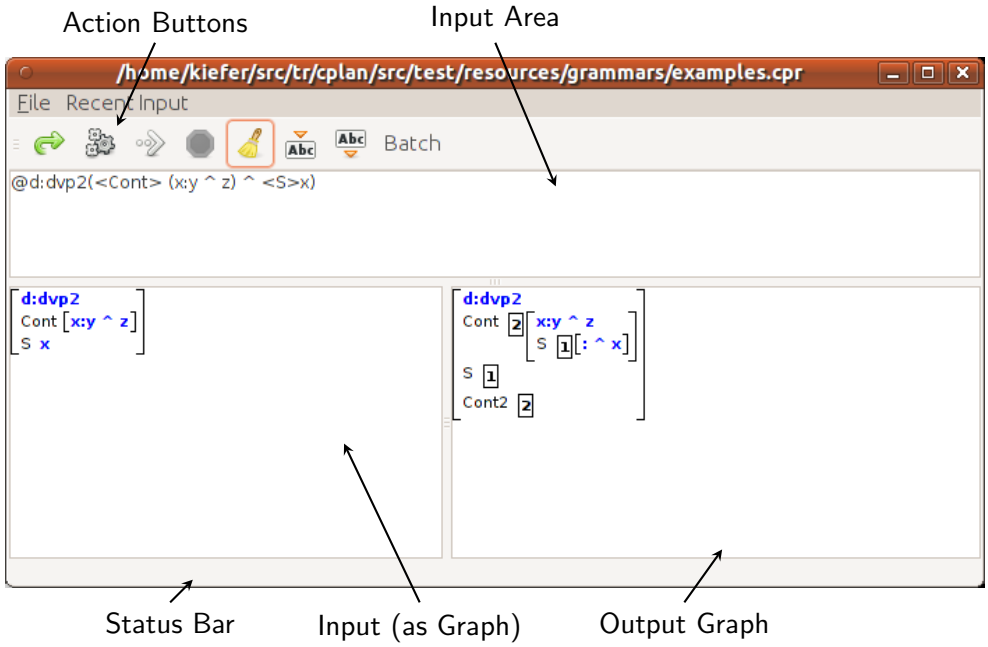

Figure 4.3: Content Planner Main Window

- Starting processing of the logical form in tracing mode, which opens a new modal trace window that is described below
- Emergency Stop in case the process is in an infinite loop
- Wipe the input area clean
- Running first the planner and then the CCG realizer for the input LF, provided a CCG grammar was given on startup

If the input contains ill-formed input, and the *Process* or *Start Trace* button has been pressed, the input area gets red and the caret will be located at the point where the error seemingly occured. In addition, the status line (at the bottom of the window) contains an error message describing the error.

4.2 Trace Window

Figure 4.4 shows the trace window that will be opened when the trace button in the main window is pressed. It allows to follow the processing of the content planner in single or larger steps, and even going backwards and forwards using the scroll bar at the bottom of the window.
The trace window consists of two main sections, the *match* section, which is in the upper half and consists of the LF window in the upper left, the bindings window in the upper right and the text window below these. This text area shows the matching part of the rule that is currently 'processed' and has been matched successfully against the structure in the window above it.

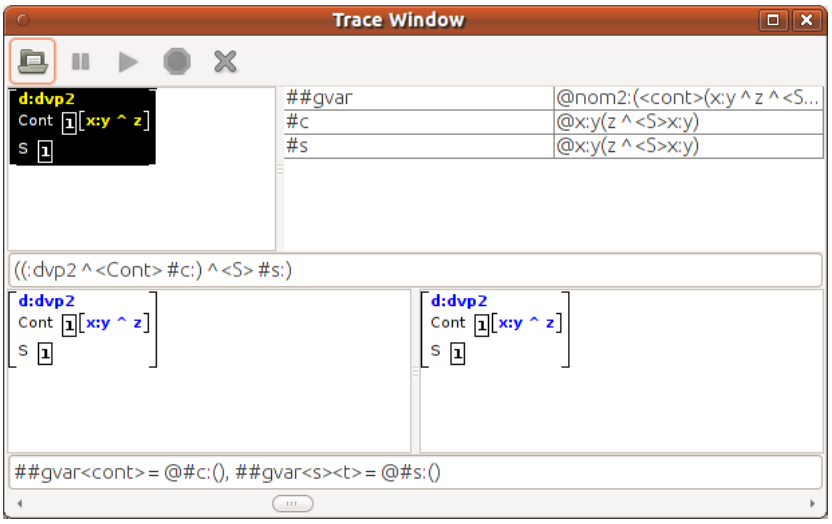

Figure 4.4: Trace Window

Since the processor first executes all rule matches pseudo-parallel to the structure and after that applies the actions one by one, this window will change not as often as the windows in the action section (see also section 9).
The second section of the trace window are the two LF windows in the bottom half and the text area for the action part of the rule below them. The lower left LF window shows the current state of the dag, before the action(s) that are shown in the text area are applied, while the lower right LF window shows the new dag after application of the action(s).
If an Emacs connection has been established, double-clicking into the match or action text area will open the appropriate rule file and jump directly to the location of the rule whose application is currently shown.
Additionally, there is a button toolbar at the top of the window, which allows you to perform the following actions:

- Jump to the current rule in the editor (the same as double clicking into the text areas)
- Suspend processing for very long or infinite runs
- Continue suspended processing
- Emergency stop (finishes processing)
- Close the trace window

4.3 Batch Test Window

Figure 4.5 shows the batch test window that will be opened when the batch button in the main window is pressed. Currently, it is designed to be used with subsequent generation. The batch files syntax is a sequence of one input form, followed by a set of possible output sentences, each on a single line, finished by an empty line. To indicate that no generation is expected, the set of output sentences may be empty. A single line with only a $*$ on it signals that any generation output is fine, except an empty output.

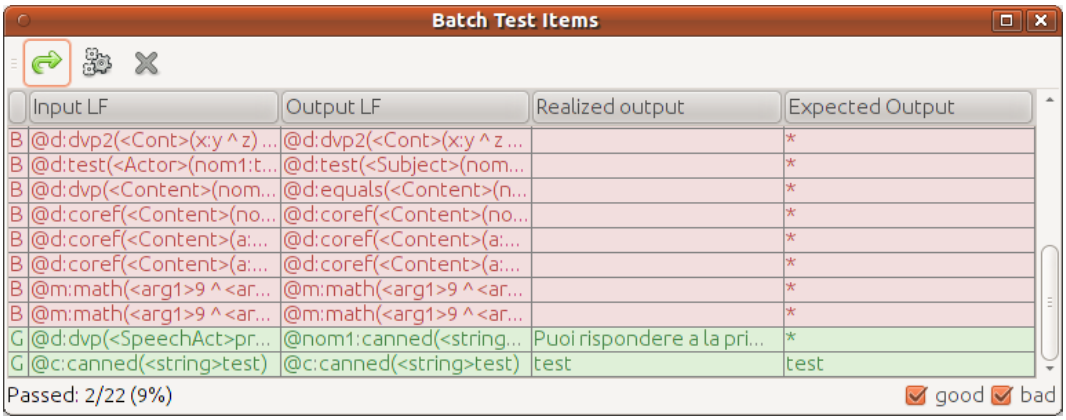

Figure 4.5: Batch Test Window

Failed test items will appear in red (or yellow, if there was a warning during processing), successful ones in green or blue. The Buttons in the tool area reload the batch file, run the batch test again or close the window, respectively.
The row header tabs of column one, four and five can be used to re-sort the table. Furthermore, the two check boxes at the bottom of the window can be used to filter the table appropriately. Clicking a row will put the input and output data into the main window, and open the corresponding line in the batch file in the editor window.

Chapter 5

General Description

The new utterance planner module is very much like a specialized graph rewriting system for logical forms as they are used in OpenCCG. A set of rules is specified by the grammar writer which is applied to every part of an input logical form, which is interpreted as a directed graph where the nominals and feature values are the nodes and the features and relations are the (named) edges.
The transfer of nodes is realized using local and global variables, which store information either during the application of a single rule, or during the whole rewrite process, respectively.

Chapter 6

Rules

A single rule has the following form:

match$^+$ *arrow* *actions* `.` with *arrow* → (`->` | `=>`) and *actions* → *action* (`,` *action*)*

When many rules share a lot of the preconditions or actions, you can use *rule groups*. The syntax for rule groups (as opposed to a single rule) is as follows:

group → *match*$^+$ (*arrow* *actions*)? ({ ((`^` | `|`) *group*) | *arrow* *actions* `.` })$^+$

All match expressions and actions are recursively collected (connecting with the given operator) until a dot is reached, and the collected actions are only applied when all the collected match conditions are fulfilled. Rules with `=>` arrows will be applied to every node as often as the match conditions are fullfilled (and the structure still changes), while rules with `->` arrows will be applied only once to a node after a successful match. In a rule group, the arrow immediately before the dot (the one that is most deeply embedded) will determine the rule type of the resulting rule.
The overall processing strategy will be explained in more detail in chapter 9.

6.1 Matching Part

The reduced BNF for one match looks like this:

match	→	*pathmatch* (`^` | `|`) *match* | *pathmatch*
pathmatch	→	`<` `ID` `>` [*pathmatch*] | *simplematch*
simplematch	→	*atomic* `:` | *atomic* [`:` *atomic*] | `!` *pathmatch* | `(` *match* `)`
		| `=` (*var* | *gvar*) `:` | `(` (*gvar* | *funcall*) `~` *match* `)`
atomic	→	`ID` | *var*
var	→	`#ID`
gvar	→	`##ID`
funcall	→	`ID` `(` *arg* (`,` *arg*)* `)` | `ID` `(` `)`
arg	→	`ID` | `STRING` | `#` | *var* | *gvar* *path* | *funcall*
path	→	(`<` `ID` `>`)*

The match part consists of atomic match expressions which can be combined with conjunction, disjunction and negation operators to obtain complex match expressions:

- `nom:` matches the nominal with name `nom`
- `:type1` matches if the node has type `type1`
- `prop` matches either the proposition value `prop`, or the value under some feature
- `<fname>` matches if `fname` either exists as feature or relation

Additionally, *local variables* also form atomic match expressions. They can bind either whole nodes, type or proposition values, and can also be used several times in a match expression to check that the value of the node is (structurally) equal to what is bound to the variable. Local variables start with a single hash (`#`), followed by a valid name. The syntax is as follows:

- `#v:` matches a whole node (nominal).
- `:#v` matches the type value of a nominal.
- `#v` matches the proposition value, or the value under a feature.
- `= #var:` checks if the node that is matched is *identical* to what is bound to the variable *var*. An unbound variable will always generate a match failure for this test.

Alternatively, a (bound) global variable can be used for the above checks. Global variables, in contrast to local variables, must begin with two hash characters, e.g., `##global_var`. Global variables can not be bound on the match side, but their values can be checked against the local value as for local variables.
There are two binary and one unary combination operator to combine atomic matches into more complex match expressions:

- `^` for conjunction
- `|` for disjunction
- `!` for negation

Parentheses can be used to group subexpressions.
There are also some short forms of conjunctions following the logical form syntax that do not require the conjunction operator, such as:

```
<feat>(nom:#type2)
```

matches a node that has a relation `feat` which points to a node with nominal name `nom` and binds the type of that nominal to the local variable `#type2`.
The match conditions, which may be empty, will always be applied to the current node that is under consideration, except if a parenthesized condition with the `~` operator is used. Then, the operand to the left must be either a global variable or a function call, and either the value bound to the variable or the return value of the function call will be matched against the match conditions to the right of the operand. If the global variable is not bound, the match is executed as if it were applied to an independent, empty node.
There is a set of build-in functions, but there is also the possibility of providing your own custom functions via plugins. For a description of the build-in functions and the plug-in interface see sections 7 and 8, respectively. For a detailed description of how the matching and modification of the dag works, see section 9.

6.2 Actions

The BNF for one action looks like this:

action	→	*appoint* `!` `<` `ID` `>` | *appoint* `=` *rexpr* | *appoint* `^` *rexpr*
rexpr	→	*rpathexpr* `^` *rexpr* | *rpathexpr*
rpathexpr	→	`<` `ID` `>` *rpathexpr* | *rsimpleexpr*
rsimpleexpr	→	*atomic* `:` [*atomic*] | *atomic* | `"STRING"` | `(` *rexpr* `)` | `ID` `(` *args* `)`
atomic	→	`ID` | *var*
var	→	`#` `ID` | `##` `ID` [`<` `ID` `>` (`,` `<` `ID` `>`)∗] | `###` `ID`
appoint	→	`#` | *var*
args	→	ϵ | *atomic* (`,` *atomic*)∗

An action consists of the following elements:

- first the specification of the place it is applied to
- an optional feature path
- an operator specifying the kind of action that is to be performed (addition, replacement, deletion)
- the structure that specifies what is added, replaced or deleted.

The place where to apply the action can be either a local or global variable, or a single hash character which represents the node against which the rule is matched.
For addition and replacement, full or partial logical forms, also containing local variables, can be specified. Deletion can currently only be applied to single features or relations.

Minimal examples for the three operations are:

Addition `# ^ <feat> val`
Add a feature / value pair to the current node

Replacement `#var = <feat> #var`
this would place the nominal that was bound to the local variable `var` in the matching part under the feature `feat` instead of its former location.

Deletion `# ! <p3>`
delete feature `p3` from the node that is currently processed

Assignment and replacement of global variables can also be used to inhibit rules, or force a certain ordering on the application of rules (see the section about matching, especially against the content of global variables). In addition, a global variable can be followed by a nonempty path of features, providing a map-like structure. On the match side, these substructures are treated like features of structured nodes and have to be extracted for binding or testing with the usual conjunction and path syntax.
In addition to the local and global variables, *right hand side local variables* can be used in multi-part action specifications. They start with three hash characters and their purpose is three fold:

- provide a means of specifying complex replacements in several steps for clarity
- create temporary structured nodes as arguments to functions.
- establish coreferences in complex nodes on the right hand side, because you are not allowed to use unbound local variables

If variables are used in the replacement of additions and assignments, no matter which kind of variables that is, you have to keep the following in mind:

- if you use `#var:`, the complete node bound to the variable will be added to the specified point of application (which can be an embedded path)
- if you use `:#var` or `#var` or `<feat>#var`, the value bound to the variable must be of a special form, because you use the content of var to specify an atomic value:
  1. Either it must be a simple (i.e., atomic) value, i.e., a variable bound to a proposition, type, or feature value, *or*
  2. It contains a complex node that has a feature with the same name as specified in the replacement, that is, a type, proposition or `feat` feature, respectively

  If the above conditions are not met, an error will be signalled.

This does not mean that you can assign a type value only to a type, you can move the atomic values around, if you do it the right way. So the following will always work (you take the type and put it under feature `x`):

```
:#var -> # ^ <x> #var.
```

This will work only if the node bound to `var` has a valid type and proposition:

```
#var: -> # ^ <y> ( :#var ^ #var )
```

6.3 Some Simple Examples

Test for a disjunction of types, and the existence of the `Shape` feature, add a new nominal with the feature's value as proposition and delete the `Shape` feature itself, to avoid infinite recursion. Since the same relation can be added multiple times, the rewriting process would never reach a fix point and, consequently, not stop.

```
(:entity | :physical | :e-substance) ^ <Shape> #v
->
# ^ <Modifier> (shape:q-shape ^ #v),
# ! <Shape>.
```

A similar example, but now, the whole node under `TopIn` is moved under the `Anchor` feature, which specifies with the parentheses and the nominal specification with the colon after the variable that it is a relation. Note that under `Modifier`, no specification of the nominal name is necessary, which results in a new, unique nominal name for that node.

```
(:entity | :physical | :e-substance) ^ <TopIn> ( #topin: )
->
# ^ <Modifier> (:m-location ^ in ^ <Anchor> ( #topin: )),
# ! <TopIn>.
```

In the next example, a local variable is used only in the action part multiple times to indicate the coreferencing of two nominals. The whole content under `Content` will be replaced due to the replacement specification for the matched local variable `c1`

```
:dvp ^ <Relation> accept ^ <Content> ( #c1:marker )
->
#c1 = (:ascription ^ <Actor>(#i1:person ^ I)
                   ^ <Patient>(:entity ^ what)
                   ^ <Subject>( #i1: )).
```

The next example adds the result of a function call to the current node. The function gets as argument the structure that is bound to the variable `c1`. A single `#` is also a valid argument to a function and represents the node that is currently processed, like in the initial place of the action specification.

```
:dvp ^ <Relation> accept ^ <Content> ( #c1:marker )
->
# ^ <Content> modify_marker(#c1).
```

Functions can also be used on the matching side (see section 6.1), as in the next example:

```
:dvp, random(one, two, three) ^ two
->
# = <Content> ( <Answer> positive ).
```

Also look into the file `presentation-examples.trf`, which contains much more elaborate examples with explanations attached to it.

Chapter 7

Build-in Functions

7.1 Mathematical Functions

Mathematical functions expecting `double` arguments will return `NaN`, if one of the arguments is not a valid number.

- `add`, `sub`, `div`, `mult` expecting two `double` arguments, returning a `double` argument, the usual binary operators
- `lt`, `gt`, `lteq`, `gteq`, `eq` expecting two `double` arguments, returning either one or zero, representing true and false, for comparison of numbers
- `not` returns zero if the (`double`) argument is non-zero, one otherwise
- `neg` is unary minus.

7.2 Other Functions

- `concatenate` concatenates an arbitrary number of strings and returns the result
- `clone` clones the argument node, to get independent copies of parts of the current structure
- `random` takes an arbitrary number of arguments and randomly returns one of them
- `throwException` takes an (optional) message argument and throws a `PlanningException` with the given message. This will end the processing of the current input.
- `warning` logs a warning message on to the `UtterancePlanner` main logger, and always returns `true`

Chapter 8

Adding New Functions

New functions can be added using a lightweight plug-in mechanism. All functions must be subclasses of `Function`, if they work on the internal structures of the planner directly, or of `LFFunction`, if they work with `LogicalForm`, as provided with the separate `CPlanWrapper` class. The methods that must be overwritten for both classes are:

```
public abstract String name();
public abstract int arity();
```

`name` must return the name as it is used in the rules, and arity the expected length of the argument list.
Additionally, the abstract method `Object apply(List args)` must be implemented for every subclass of `Function`, except for `LFFunction`, where `LogicalForm applyLfFunction(List args)` must be implemented instead. The length of args matches the arity specified by the function.
If the functions are not directly compiled in and registered at the `FuncctionFactory`, they can be provided as plug-ins. The class files of the functions have to be added to a jar-archive such that the path where they are stored in the archive matches the package prefix.

```
$ jar tf plugin-jars/cplan-plugins.jar

META-INF/
META-INF/MANIFEST.MF
de/dfki/lt/tr/dialogue/cplanwrapper/ChangePropFunction.class
```

Figure 8.1: `de.dfki.lt.tr.dialogue.cplanwrapper.ChangePropFunction` properly put into a `.jar` file to use as plug-in function

These jars have to be put into one directory, which is passed to the constructor of the `UtterancePlanner` class as second argument. The constructor will then load all appropriate classes (subclasses of `Function`) on startup.

Chapter 9

Processing Regime

Processing is divided into different stages. Every stage has its own set of rules, which are loaded from the rule files specified in different sections of the project file (see 3 for further information), and essentially behaves in the same way.
For every stage, the current processing engine proceeds as follows:

- Traverse the structure in preorder sequence, starting at the root node
- At every node, match all rules against the node in the order in which they have been loaded

  If the match was successful, and either it is a `=>` rule or the rule has not been applied to this node before:

  store the rule together with the local bindings of the match for later execution of the rule's actions

- Execute the actions for the stored rule / binding pairs in the order in which they were stored, which is the order in which the matches occured.

Since the matches are applied before any changes are made, any applicable match will be found. When the changes are applied afterwards, is is possible that some of them will not have any effect on the input structure, say, if something is added to a node that has previously been deleted.
This process is iterated until a fixpoint is reached, i.e., the structure does not change by applying the rules as described. If the current stage was not the last stage, the structure is then processed with the next set of rules, until all stages have been applied. Every stage is only applied once, there is no fixpoint loop around the stages. The stages are applied in the order in which they appear in the project file, independent of their names.

Appendix A

More Examples

The following examples are a LaTeXed version of the file `examples.trf`, which is used also for testing and can be found in the directory `core/src/test/resources/grammars` to try out the things in the running system, and to experiment with variations.

Example 1: A sub-node is replaced with fresh content. Notice the match condition for the absence of a feature.

```
/// @d:dvp(<SA>assertion ^ <Rel> accept ^ <Cont> (:conttype ^ <foo> bar))
:dvp ^ ! <Ackno> ^ <SA> assertion ^ <Rel> accept ^ <Cont> ( #c1: )
->
#c1 = (:marker ^ ok).
```

Example 2: Selecting a specific relation, and adding to it

```
/// @aa:bb(<C>(<Mod>(:g ^ <x> y)))
<C>(<Mod>(#m:g ^ ! <Cont>)) -> #m ^ <Cont>(:new ^ clean).
```

Example 3: Add a default in case of feature absence

```
/// @aa:ascr(<C>d)
:ascr ^ !<Tense> -> # ^ <Tense>pres.
```

Example 4: Set a global variable as marker, and test in the second rule

```
/// @d:dvp(<SpeechAct>assertion ^ <w>(:foo ^ <Tense>pres))
:dvp ^ <SpeechAct>#v -> ##speechact = #v.

:foo ^ <Tense> ^ (##speechact ~ assertion) -> # ^ <Mood>ind.
```

Example 5: type disjunction, variable `t` matching the whole node add two relations, delete `Target` feature (multiple actions)

```
/// @d:disj(<T>(:entity ^ <Tense>pres))
/// @d:disj(<T>(:thing ^ <Tense>pres))
:disj ^ <T> #t:(entity | thing)
->
# ^ <CR>(#t:) ^ <Subject>(#t:),
# ! <T>.
```

Example 6: Less preferable rewrite of the last example, the same variable name has to be used twice! Works only because of disjunction.

```
/// @d:disj2(<T>(:entity ^ <Tense>pres))
/// @d:disj2(<T>(:thing ^ <Tense>pres))
:disj2 ^ (<T> (#t:entity) | <T> (#t:thing))
->
# ^ <CR>(#t:), # ^ <Subject>(#t:),
# ! <T>.
```

Example 7: Using `=>` rules to walk through a list

```
///@d:dvp(<c>0 ^ <l>(:li ^ <f>1 ^ <r>(:li ^ <f>2 ^ <r>(:li ^ <f>3 ^ <r>()))))
///will become: @d:dvp(<c>0123 ^ <l>(:li))

<c>#c: ^ <l>(#l: ^ <f>#f ^ <r>#r:)
=>
#l = #r:,
#c = concatenate(#c, #f).
```

Example 8: Rule groups

```
:dvp ^ <SpeechAct>greeting
^ <Context>(<Familiarity>yes ^ <Encounter>notfirst ^ <Expressive>yes)
->
###greet = random("ciao ", "buongiorno ", "ehi , ciao "),
###polite = random(", come stai ? ", "! "),
###init=concatenate(###greet, ###polite)
###pleasure1 = "sono contento di rivederti ",
###pleasure2 = "sono felice di rivederti ",
# ^ :canned ^ <SpeechModus>indicative
{
^ <Context>(!<ChildGender> | <ChildGender>unknown)
->
# ^ <stringOutput>concatenate(###init, random(###pleasure1, ###pleasure2)).

^ <Context>(<ChildGender>m)
->
# ^ <stringOutput>concatenate(###init,
                    random(###pleasure1, ###pleasure2, "bentornato ")).

^ <Context>(<ChildGender>f)
->
# ^ <stringOutput>concatenate(###init,
                    random(###pleasure1, ###pleasure2, "bentornata ")).
}
```

Example 9: Randomizing with complex values

```
/// @d:rand(<SpeechAct>opening ^ <Content>(:top ^ <X> y))
:rand ^ <SpeechAct>opening ^ <Content> (#c1:top)
->
###opening1 = :opening ^ "hi, dude",
###opening2 = :opening ^ hello,
###opening3 = :opening ^ "nice to see you" ^ <form> polite,
# ! <SpeechAct>,
/// Note the colon after the function call! It means that the whole node is the
/// value, not just the proposition.
#c1 = random(###opening1, ###opening2, ###opening3): .

/// @d:rand2(<SpeechAct>opening ^ <Content>(:top ^ <X> y))
:rand2 ^ <SpeechAct>opening ^ <Content> (#c1:top)
->
###opening1 = "hi, dude",
###opening2 = hello,
###opening3 = "nice to see you" ^ <form> polite,
# ! <SpeechAct>,
#c1 = random(###opening1, ###opening2, ###opening3):opening .
```

Example 10: Alternative randomization, maybe not very convenient

```
/// @d:dvp(<SpeechAct>closing ^ <Content> (foo))
:dvp ^ <SpeechAct>closing -> ##randomclosing = random(1,2).

:dvp ^ <SpeechAct>closing ^ <Content> (#c1:) ^
(##randomclosing ~ 1)
->
#c1 = :closing ^ bye.

:dvp ^ <SpeechAct>closing ^ <Content> (#c1:) ^
(##randomclosing ~ 2)
->
#c1 = :closing ^ see_you.
```

Example 11: Using global variable as node store.
After application of these rules `Target` and `PointToTarget` point to the same node.

```
/// @d:dvp(<Speechact>assertion ^ <Content>(a:ascription))
:ascription ^ #t: -> ##fromStore = #t:.

:ascription ^ ! <PointToTarget>
->
Again, note the colon after the global variable in the addition
# ^ <PointToTarget> ##fromStore:.
```

A.0.1 Adding to relations the wrong way

```
/* Test input
@a:dvp(foo ^ <SpeechAct>question ^
             <Content>(c1:ascription ^
                 <Subject>(s1:entity ^ <Delimitation>unique) ^
                 <Cop-Scope>(s2:gaga ^ prop ^ <Questioned>true)))
 */
```

Example 12: This will not add to the existing `Subject` and `Cop-Scope`, but introduce new ones.

```
:dvp ^ <SpeechAct>question
     ^ <Content>(#cont:ascription ^
                 <Subject>(:entity ^ <Delimitation>unique) ^
                 <Cop-Scope>(#cop-scope: ^ <Questioned>true))
->
#cont ^ <Wh-Restr>(:specifier ^ what ^ <Scope> #cop-scope:)
      ^ <Subject>( context ^ <Proximity> proximal )
      ^ <Cop-Scope>(<Delimitation>unique ^ <Num> sg),
#cop-scope ! <Questioned>.
```

Adding to relations: the correct alternative:
This adds the new information to the previously matched nodes.

```
:dvp ^ <SpeechAct>question
     ^ <Content>(#cont:ascription ^
                 <Subject>(#subj:entity ^ <Delimitation>unique) ^
                 <Cop-Scope>(#cop-scope: ^ <Questioned>true))
->
#cont ^ <Wh-Restr>(:specifier ^ what ^ <Scope> #cop-scope:),
#subj ^ context ^ <Proximity> proximal,
#cop-scope ! <Questioned>,
#cop-scope ^ <Delimitation>unique ^ <Num> sg.
```

A.0.2 Global variable "maps"

Example 13: Global variables can be used with path expressions to get dictionary-like structures

```
/// @d:dvp2(<Cont> (x:y ^ z) ^ <S>x)
:dvp2 ^ <Cont> #c: ^ <S>#s: -> ##gvar<cont> = #c:, ##gvar<s><t> = #s:.

:dvp2 ^ !<SpeechAct> ^ !<Cont2> ^ ( ##gvar ~ <cont> #v: )
->
# ^ <Cont2> ( #v: ^ <S> ##gvar<s><t>: ).

/// or, alternatively for the second rule (uncomment to test)
/*
:dvp2 ^ !<SpeechAct> ^ !<Cont2> ^ ( ##gvar ~ <cont> )
->
# ^ <Cont2> ( ##gvar<cont>: ^ <S> ##gvar<s><t>:).
*/
```

A.0.3 All you can do with variables

Example 14: Bind values to global variables and use them in another rule

```
/// @d:dvp(<foo>(:a ^ <F>(b:c ^ d)))
:a ^ <F> (#i:#t ^ #p) -> ##partial = :#t ^ #p, ##whole = #i:.

:a ^ !<W> -> # ^ <W> ##whole: ^ <P> ##partial:.
```

Example 15: Be careful that you use bound variables correctly! If you use them as simple (type or proposition) values on the right hand side, you must have bound them to simple values, or complex values containing the appropriate edge!
The second rule illustrates how to get type and prop out of a complex node in a global variable

```
/// @d:test(<Actor>(:type ^ prop ^ <foo> bar))
<Actor>(#a:) -> ##s = #a:.

:test ^ ! <Subject> ^ (##s ~ (:#type ^ #prop))
->
# ^ <Subject> ##s: ^ <Prop> #prop ^ <Type> :#type.
```

Right hand local variables can also be used to establish coreferences in the replacement part.

Example 16: Check for structural equality (will also succeed if coreferent)

```
/// @d:dvp(<Content>(:bar ^ baz) ^ <Wh-Restr>(:bar ^ baz))
:dvp ^ <Content>#c: ^ <Wh-Restr>#c: -> # ^ :equals.
```

Example 17: Check for identity (will only succeed if coreferent)

```
/// @d:coref(<Content>(:bar ^ baz) ^ <Wh-Restr>(:bar ^ baz))
/// @d:coref(<Content>(a:bar ^ baz) ^ <Wh-Restr>(a:bar))
/// @d:coref(<Content>(a:bar ^ baz) ^ <Wh-Restr>a:)
:coref ^ <Content>#c: ^ <Wh-Restr> = #c: -> # ^ identical.
```

A.0.4 Use of functions for tests and results

Example 18: Bind values to global variables and use them in another rule

```
/// @d:dvp(<SpeechAct>provideQuestion ^ <Context>(<Question> "question " ^ <Count> "1" ))
:dvp ^ <SpeechAct>provideQuestion
     ^ <Context>(<Question> #q ^ <Count> #x: ) ^
( eq(#x, 1) ~ 1 )
->
###part1 = random("La prima domanda è: ",
                  "Ecco la prima domanda: ",
                  "Qui viene la prima domanda: ",
                  "Puoi rispondere a la prima domanda: "),
# = :canned ^ <string>concatenate(###part1, #q, "?").
```

Example 19: Non-integer numbers must be passed to functions as strings. Boolean functions return zero or one.

```
/// @m:math(<arg1>9 ^ <arg2>2)
/// @m:math(<arg1>9 ^ <arg2>"30.33")

:math ^ <arg1>#arg1 ^ <arg2>#arg2 ^ (lteq("0.3", div(#arg1, #arg2)) ~ 1)
->
# ^ <res> div(#arg1, #arg2).

:math ^ <arg1>#arg1 ^ <arg2>#arg2
->
# ^ <bool> lteq("0.3", div(#arg1, #arg2)).
```

Example 20: You can use arbitrary encodings in your grammar files, but if it's not UTF-8, you have to specify as 'encoding' setting in the grammar file, and, consequently, you can only have one encoding per project.

```
/// @e:enc(<enc> iso-8859-15 ^ <val> "äÖìâé")
<enc> iso-8859-15 ^ <val> "äÖìâé" -> # ^ <right> true.
```