lc-parse-gui - A Left-Corner Parser with Graphical Debugger

Seminar für Sprachwissenschaft
Eberhard Karls Universität Tübingen

Author:
Johannes Dellert
johannes.dellert@gmx.de

Course:
Logic Programming

Supervisor:
Dr. Frank Richter

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln (einschließlich des WWW und anderer elektronischer Quellen) angefertigt habe. Alle Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht.

(Johannes Dellert)

Contents

1 Introduction

This paper documents `lc-parse-gui`, an experimental parsing environment that combines a left-corner parser in plain Prolog with a graphical debugger written in Java.

The graphical debugger allows the user to manually guide the parsing process by confirming or rejecting hypotheses, and it provides the user with the possibility to inspect in detail the state of the parser at each step of the parsing history.

Left-corner parsing was developed as one of the standard approaches to parsing context-free grammars (see Irons (1961) for the seminal paper, and Rosenkrantz & Lewis (1970) for a more explicit discussion). It can be seen as a combination of top-down and bottom-up parsing in that it is input-driven, but relies on top-down predictions to maintain a hypothesis about the parse tree.

In principle, the parser scans a word from the input, makes predictions about which structure the scanned word could be part of, and tries to complete the predicted structure. This procedure is repeated recursively for non-leaves by means of a prediction stack in which incomplete predictions are stored.

Left-corner parsers have become increasingly popular in psycholinguistics because there are strong indications (see e.g. Resnik (1992)) that of all the classical parsing approaches, left-corner parsing resembles most the way humans process language. This is plausible because of the linearity with which a left-corner parser infers as much as it can about the parse tree without knowing all the input.

The benefits of a parser with a graphical debugger like the present one are twofold: On the one hand, it can be very useful for researchers in psycholinguistics to be able to manually guide parsing processes towards realistic processing in order to recognize which decision patterns will have to be contained in their models of sentence processing. On the other hand, it can be a valuable tool for teaching the basics of left-corner parsing to students, since it is always easier to understand new algorithms with the support of an intuitive visualisation.

The basic structure of the Prolog part of the software relies on the left-corner parser implementation presented in Covington (1994) in section 6.4.1. This backbone had to be severely extended to provide support for explicit prediction stack maintenance, construction of the decision tree, and message exchange with the GUI.

The communication between Prolog backend and Java frontend relies on the Jasper library whose functionality is best explained in the SICStus documentation (SICStus (2007), chapter 43).

2 Installation and Basic Usage

2.1 Requirements

Basic system requirements:

- SICStus Prolog 3.12.8 or higher, not tested with SICStus 4
- Sun Java Version 1.6.0.12 or higher

2.2 Installation and Usage

To be able to display the user interface of `lc-parse-gui`, SICStus must be installed with Jasper support. If you installed SICStus on your system without activating Jasper support, the easiest way under Linux is to reinstall SICStus by means of the install script `InstallSICStus` that comes with it. When prompted, state that you want to install Jasper, and specify the location of your Java installation. Normally, this would be the directory in which the `bin` folder with the `java` executable resides (check with `which java` if unsure).

If you have multiple version of Java installed, make sure you point Jasper to the correct one. Especially note that the `gcj` Java variant contained in some Linux distributions is not compatible with Jasper. If you must use an older version of Sun Java 6, you might be successful by recompiling the Java files contained in the `src` folder, and moving the resulting Java classes to the `bin` folder.

Once SICStus and Java are correctly set up, installing `lc-parse-gui` merely amounts to unpacking the contents of the lc-parse-gui archive into a directory of your choice.

To use the software, change into the installation directory and fire up `sicstus` from there. At the prolog prompt, simply load the parsing system by telling Prolog to consult `lc-parse-gui.pl`:

```
?- ['lc-parse-gui.pl'].
```

The next step is to load a grammar. This is also achieved simply by consulting the Prolog file containing the grammar. In the case of the English test grammar that comes with lc-parse-gui, this amounts to typing

```
?- ['test-grammar-en.pl'].
```

The system is now ready to parse sentences by means of the `parse/2` predicate. With the test grammar, we could tell lc-parse-gui to try to analyze "the dog chases the cat" as something of category `s`, or to parse "amuses the cats near the elephant" as a `vp`:

```
?- parse(s,[the,dog,chases,the,cat]).
?- parse(vp,[amuses,the,cats,near,the,elephant]).
```

A graphical user interface will appear, allowing the user to steer and inspect the parsing process. The usage of the interface is described in detail in Chapter 4.

3 Grammar Format

`lc-parse-gui` operates on standard CFGs in a Prolog format.

3.1 File Format

The input format for the parser are Prolog files that contain clauses of the predicates `rule/2` and `word/2`. The first argument of both predicates contain the category whose internal structure the clauses define, or the left-hand sides of production rules. For clauses of `rule/2`, the right-hand side must be a list of categories representing the right-hand side of the production rule. In the case of `word/2`, the second argument is a prolog atom representing a word in the described language.

Consider e.g. the following clauses from the English sample grammar:

```
rule(np,[d,n]).

word(d,the).
word(n,dog).
word(n,dogs).
```

This fragment defines that "the" can be parsed as a `d`, while "dog" and "dogs" are of category `n`. The rule states that an `np` can be realized as a `d` followed by an `n`. Taken together, the fragment licenses two phrases of type `np`: "the dog" and "the dogs".

Note that it is not a problem to assign two different categories to one word by using two `word/2` clauses with an identical second argument. The parser will backtrack to take both possibilities of category assignment into consideration.

3.2 Limitations

The parser currently cannot handle null constituents, i.e. setting the second argument of some `rule/2` clause to the empty list will lead to unpredictable behaviour. In such a case, the parser will either crash or descend into infinite recursion.

To understand why this happens, consider the following rules:

```
rule(s,[np,vp]).
rule(np,[d,n]).
rule(d,[]).
```

Note that these rules are supposed to license an empty determiner at the start of an `np` and an `s`. The problem is that when the parser is looking for a `vp`, but the `vp` is not there, it will predict an `np` headed by an empty determiner instead. After not finding an `n` to complete this, it will predict another `np` with an empty determiner, and so on ad infinitum.

It is possible to overcome such problems by precompiling the possible beginnings of constituents when loading the grammar and to only allow the prediction of

an empty constituent if the symbol that is being completed at the moment can begin with that constituent. Such a pre-compilation step is planned for future versions of the parser. The main reason why this feature was not integrated into this prototype is that the parsing process would have been harder to visualize.

In many cases, empty constituents are not necessary to ensure the productivity of the grammar, even though the analyses obtained without empty constituents might violate principles of linguistic theory. In the example case, the same strings would be described by the grammar if we simply changed the rules into

```
rule(s,[np,vp]).
rule(np,[d,n]).
rule(np,[n]).
```

4 Graphical Debugger

The graphical debugger provides the user with the possibility to influence the alternatives considered by the parser. What is technically not more than what one could do with a standard Prolog tracer becomes more useful through explicit visualisation of the decisions the parser or the user made during the process. The interface allows the user to inspect the structure predicted by the parser after each step, but there is also the possibility of inspecting the structure at previous stages because all the data on each step are stored and made browsable by the debugger.

Figure 1 shows a typical state of the GUI after some parsing steps. The interactive decision tree visualizes each step or decision point by a node, and it branches out if the parser backtracked to a previous decision point. By clicking on nodes of the decision tree, the user may inspect the parser's state at each point in history. The hypothetical tree maintained by the parser at the currently selected decision point is visualized in a panel below the decision tree. A parsing status message at the bottom of the window explains verbally the last step the parser performed to arrive at the current structure hypothesis. By means of the five buttons on the control panel, the user may steer the parsing process by confirming or rejecting hypothetical structures, influencing the backtracking behaviour of the Prolog parser.

All the components of the GUI and their interactions are described in detail in the following sections.

4.1 Controlling the Parsing Process

The graphical debugger allows for the user to steer the parsing process by means of the five buttons in the control panel.

Continue: This simply allows the parser to continue to the next step, no steering directive involved. Continuously clicking this button will cause the parser to incrementally explore the entire structure search space and to try out all alternatives until the first parse is found. Then, the parser will stop. The behaviour of `Continue` is equivalent to the "creep" directive in a standard Prolog tracer.

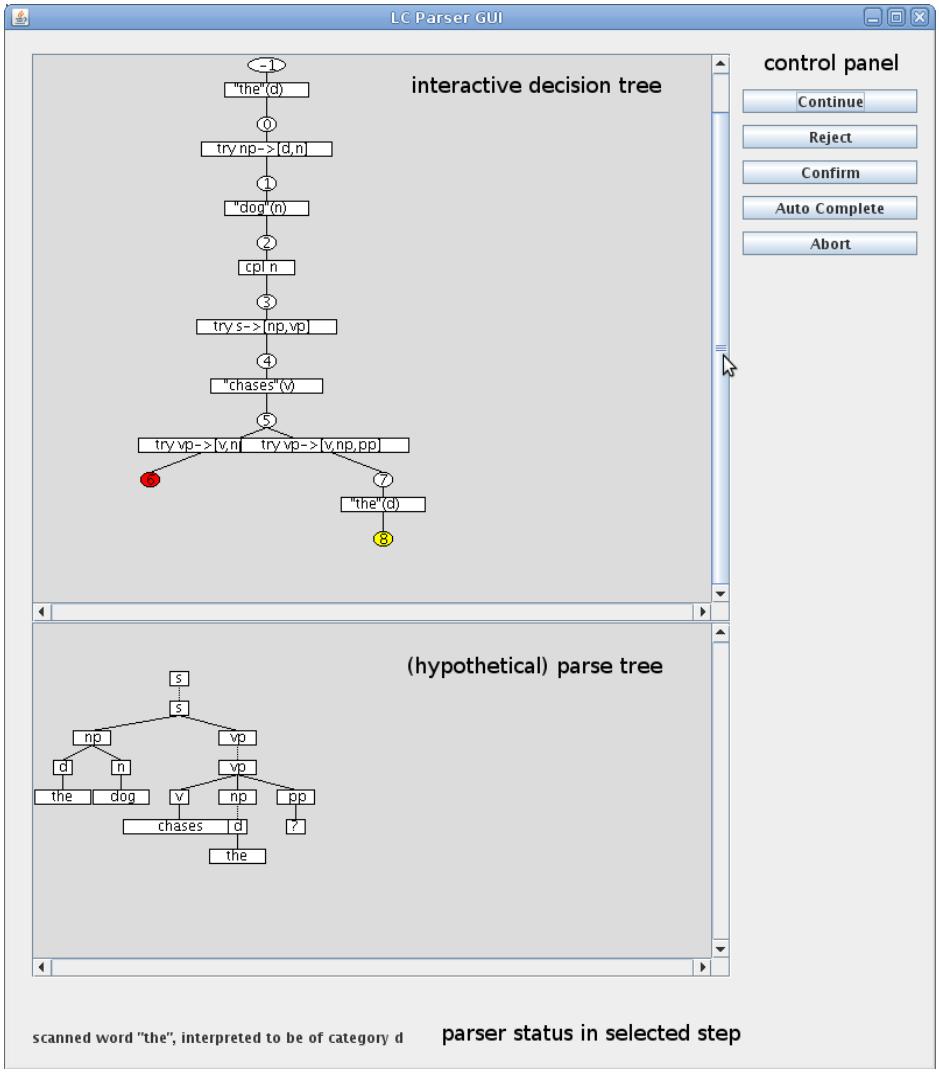

Figure 1: The graphical debugger and its components

Reject: This serves to reject the current structure hypothesis, causing the parser to fail on the current branch of the decision tree. All variants that build on the rejected hypothesis will not be explored. Can be used to prune predictions that will not lead anywhere, or to block the solution usually found in order to explore alternative parses. The behaviour of `Reject` is equivalent to the "fail" directive in a standard Prolog tracer.

Confirm: Tells the parser to only explore possible parses that build on the currently predicted structure. If the parser fails to find such a parse, it will not backtrack beyond this point and try to find a parse starting from previous hypotheses, but it will fail. Can be used to explicitly avoid many branches that would otherwise have to be cut using `Reject` to achieve the same effect. On the Prolog side, this corresponds to introducing a cut into the currently evaluated predicate. There is no equivalent to this in a standard Prolog tracer.

Auto-Complete: Tells the parser to automatically `Continue` as fast as possible. This will cause the decision tree to grow very fast, and is very useful for jumping over uninteresting parts of the parsing process without having to click on the `Continue` button many times. Auto-complete mode can always be cancelled by clicking on one of the other buttons. This can be used to approximate the behaviour of the "skip" directive in a standard Prolog tracer, with the advantage that skipped steps can still be investigated because their details are also handed on to the GUI.

Abort: Causes the parsing process to be aborted. Very useful if two parses are to be compared, because this will leave the debugger window open while at the same time aborting the Prolog query to make the Prolog console free for user input again. The user can then recompile the grammar or start another parse and thus graphically compare the actions of the parser on different inputs or with a modified grammar.

4.2 Interpreting the Decision Tree

This section explains the semantics of the decision tree, what the different edge labels and node colors mean, and how this information can be used to find bugs in a parser.

The basic idea of the decision tree visualization is to make the parsing process as transparent as possible by showing to the user how the parser backtracks to explore all possible solutions. Whenever a new structure hypothesis is reached by performing a parse step starting from the hypothesis associated with some node in the decision tree, that node will spawn a new child node. This means that an entirely deterministic parser that does not have to backtrack will have just one long branch as its decision tree. If at some point the parser has multiple choices to consider, sooner or later the decision tree will branch at that point, provided that the parser does not already find a parse resulting from the first choice. The nodes of the tree are numbered in order of generation, allowing the user to see even at later stages in which order the tree was constructed.

In principle, there are three kinds of steps the parser can perform at each point: it can scan input symbols, predict new structure and complete predicted structures. Those three alternatives are mirrored by the labels of the decision tree edges, providing information on which step allowed the transition from one parser state to the next:

"WORD"(CAT): Edge labels of this format mean that the input symbol *word* was scanned and interpreted to be of category `cat`. The partial parse tree will be enlarged by a dangling leave node, and in the next step the parser will try to integrate this new node into its structure hypothesis.

try LHS → RHS: This means that a new substructure was predicted to accomodate the current dangling leaf by means of the phrase structure rule `rule(LHS,RHS)`. Usually, the first symbol on the RHS of the predicted substructure will be used to attach the leaf, and now the substructure has to be completed by filling in structures for the other parts of the RHS.

cpl CAT: The parser has managed to complete one of the gaps in the hypothetical parse tree by recogizing that a dangling leaf of category CAT can be linked to an incomplete predicted branch further up in the structure.

Some of the nodes in the decision tree are colored, with the following intended meanings:

Red color marks nodes in the decision tree at which the user pressed the `Reject` button, causing the parser to not consider branches starting at this node any longer. This visualizes the direct correspondence between rejection of structure hypotheses and cutting off decision tree branches. If a parse unexpectedly failed, the user should check back on all the red nodes whether all the structure rejections were justified.

Green color marks nodes in the decision tree at which the user pressed the `Confirm` button, causing the parser to not spawn any more decision tree branches above this node. The cut introduced by structure confirmation thus kills all the branches above it, limiting the growth of the decision tree to the part below the cut. In case of unexpected parse failure, the user should also check that the expected parse could really have been generated from the hypothesis confirmed at the green node deepest down in the tree.

Yellow color marks the currently selected decision tree node. The structure hypothesis the parser had at this stage is displayed in the parse tree view, and a more verbose description of the step leading to that structure is displayed in the parser status display.

4.3 Exploring the Parsing History

The main advantage of the graphical debugger is the possibility to inspect the structure hypothesis of the parser at each point in history. A click on a node in the decision tree will select that node and display the information the parser had at that stage in the bottom section of the screen.

The parse tree hypothesis displayed is in essence an incomplete parse tree representing the knowledge (or assumptions) of the parser about the sentence structure at a given point. This hypothetical structure might include gaps where the parser is still unsure how to link parts of the tree, delaying the decision which intervening structure will link two tree fragments in the end. In the parse tree display, such a situation is represented by a dotted line.

The parser status message displayed under the parse tree hypothesis once more expresses in a more verbose manner what is already displayed on the decision tree edge leading to the currectly selected node. The status message can thus be seen as a verbal description of the last change to the parse tree hypothesis. This is also the place where the message about the parser's success will appear after completion of the parsing process.

5 Technical Details

This chapter is primarily intended for readers with technical interest in the internals of the system. Some of the more interesting points about the parser's architecture are explained and motivated.

The first section features an overview of the core parser's Prolog implementation. Some important predicates, their tasks and their interaction are explained. The second section explains in detail how Jasper was used to let the Prolog process spawn a Java Swing window, and how information exchange between Prolog backend and Prolog frontend is achieved. The last section briefly comments on the design of the graphical debugger, focussing on data structures and decision tree construction.

5.1 Parser Architecture

The core of `lc-parser-gui` are the predicates whose interaction steers the parsing process. This task is fulfilled by the four predicates `parse/2`, `parse_word/8`, `parse_list/8`, and `complete/9`. The tasks fulfilled by each of these predicates are as follows:

parse/2 provides the user with the interface to start a parse. Its signature is `parse(+Category,+Sentence)`, where `Category` is a category as used on the left-hand side of rules in the grammar, and `Sentence` is a list of atoms representing the tokens of a sentence. This predicate opens the GUI window (more on this later), initializes an internal structure called the *prediction stack* that represents the parse tree hypothesis, and calls `parse_word/8` to perform the parse.

parse_word/8 essentially performs a scan step, removing the next symbol from the input list and looking up its category in the grammar. Then it calls `complete/9` to integrate the new leaf into the parse tree.
The signature of this predicate is `parse_word(+C,+InputList,-NewInputList, +PdctStack,-NewPdctStack, +JavaEnv, +StackTrace, -FinalStackTrace)`,

where `C` is the goal category, `InputList` and `NewInputList` as well as `PdctStack` and `NewPdctList` are the input lists and the prediction stacks before and after execution. `JavaEnv` contains the Java environment needed to transmit the step information to the GUI, see the next section for more information. `StackTrace` and `FinalStackTrace` are lists of numerical IDs that are used to determine the position in the decision tree before and after execution of the predicate.

parse_list/8 is called after a prediction step to complete missing symbols on the right-hand side of the rule used for the prediction. It recurses down the list of missing symbols, calling `parse_word/8` once at each step to find material that can fill the holes in the structure.
The signature is `parse_list(+CatList,+InputList,-NewInputList, +PdctStack,-NewPdctStack,+JavaEnv,+StackTrace,-NewStackTrace)`, with the same meanings as for `parse_word/8`, except that the first argument is now a list of categories that still have to be processed to complete the current prediction.

complete/9 serves two different purposes in trying to integrate newly parsed symbols into the structure. If the symbol currently processed fits into a hole in the prediction stack, `complete/9` will plug the two matching tree fragments together and thus reduce the prediction stack (the `cpl` case in the GUI). If the current symbol does not fit into any hole, `complete/9` will predict intermediate structure that might be integrated later, enlarging the prediction stack (rule application case in the GUI). In the latter case, the new structure will introduce a new list of unmatched symbols that have to be filled using `parse_list/8` before the new structure can be linked to the prediction stack by means of a recursive call to `complete/9`.

The internal format for the prediction stack is a list of pairs of prolog lists. Each entry in the list corresponds to one level of structure that has not yet been linked to structure at higher levels. In the visualisation, the tree fragments contained in different levels of the prediction stack are linked by dotted lines. An entry in the stack is a Prolog list of length 2, where the first entry of the list contains a tree fragment encoded as a nested list, and the second entry is a list representing the current holes in the structure. Each entry in the hole list is bound to a hole location in the structure and contains a variable that allows for comparatively easy structure plugging by unification.

5.2 Java-Prolog communication

By means of the Jasper library, it is possible to handle a Java Virtual Machine as an object bound to a Prolog variable. Using this object, the Jasper library can create instances of Java classes, and this includes classes inheriting from `JFrame` that represent GUI windows in Java's Swing library. The JVM object together with the instantiated GUI class are handed on to all the predicates during the parsing process as a pair contained in the `JavaEnv` variable.

All the methods of the GUI class that are to be called by Prolog must be declared beforehand. This is achieved by declaring clauses of the predicate `foreign/3`, the arguments being a clause of `method/3` that contains informa-

tion on the Java name of the predicate, the atom `java` and the prolog signature of the predicate (with the Java object as its first argument) that will correspond to the external method, e.g.

```
foreign(method('LCParserGUI','parsingFailure',[instance]),
        java,parsing_failure(+object('LCParserGUI'))).
```

The calling of such predicates is wrapped into the predicate `call_foreign_meta/2`, which is defined in the source code by the following clause:

```
call_foreign_meta(JVM, Goal) :-
   functor(Goal, Name, Arity),  % extract predicate name
   functor(ArgDesc, Name, Arity), % build template
   foreign(Method, java, ArgDesc), % look it up
   !,
   jasper_call(JVM, Method, ArgDesc, Goal).
```

This predicate is used by helper predicates used suring parsing to send messages to the frontend, as for example

```
notify_failure([LCParserGUI,JVM], Result) :-
    call_foreign_meta(JVM,parsing_failure(LCParserGUI)).
```

In the same way, the parsing predicates transmit the information that they have been called to the GUI along with the current states of the prediction stack and the call stack. For each of these purposes, the `LCParserGUI` class has a specialized method that processes these pieces of information to build a model of the parsing process that it allows the user to explore.

In the other direction, information on the buttons clicked by the user on the frontend has to be transmitted back to Prolog. This is implemented in a classical polling pattern. In principle, whenever one parsing step is completed, the Prolog backend enters a loop, calling ten times a second a specialized method of the `LCParserGUI` class that returns the ID of the clicked button. As long as the user has not clicked on any button, the ID `none` is returned and the loop continues. As soon as teh user clicks a button, a field of the Java class changes its value, causing the method to return a value that the parser can react to. This is how the loop is implemented:

```
await_gui_guidance(JavaEnv, Pressed) :-
    wait_until_button_pressed(JavaEnv,Pressed), !.

wait_until_button_pressed(JavaEnv,Pressed) :-
    repeat,
    sleep(0.1),
    pressed_button(JavaEnv,Pressed),
    ((Pressed == 'confirm');
    (Pressed == 'continue');
    (Pressed == 'reject');
    (Pressed == 'auto_complete');
    (Pressed == 'abort');
    (Pressed == 'close_window')).
```

The following block of Prolog code is part of the definition of each parsing predicate and determines the parser's reactions to the different possibilities of user input:

```
await_gui_guidance(JavaEnv, Signal),
(
 (
  Signal == 'abort',
  abort
 );
 (
  Signal == 'close_window',
  terminate_gui_execution(JavaEnv),
  abort
 );
 (
  Signal == 'reject',
  fail
 );
 (
  Signal == 'confirm',
  !
 );
 (
  Signal == 'continue'
 );
 (
  Signal == 'auto_complete'
 )
)
```

This communication method is rather limited since it is only useful for transporting small bits of information back to the Prolog side at very precisely defined points, but it has the advantage of circumventing any need for communication via ports or sockets.

5.3 Graphical Debugger

The Java classes for the graphical debugger are structured in a standard manner, without any surprising technological choices. The only point that needs to be explained here is how the model of the parsing process is built from the information that comes in from Prolog calling the methods designed for that purpose. This touches upon the internal data structures for parsing history handling as well as upon the conversion of the data the Prolog side provides.

Information about each parsing step and its corresponding decision tree nodes is transmitted by calling `addParseStepToHistory()` with four string arguments:

stackState contains a linearization of the current prediction stack state in Prolog. The tree hypothesis to be displayed by the GUI is already assembled under Prolog using unification, all nodes that are to be linked to their parent

nodes by dotted lines are prefixed with a star symbol. When the linearization is parsed by the debugger to form a tree structure that can be displayed, this markup symbol is converted into the instruction to use a dotted line for the line from the current node to the parent.

stepDescription is the status message that will later be displayed under the tree hypothesis visualization. It can directly be taken over as a string.

shortDescription is the label for the edge linking the current parsing state to its parent in the decision tree. This will be added to the decision tree model.

callTrace is a list of integers that determines the location of the new node in the tree. After parsing this into a Java array of integers, the first entry is interpreted as the new decision tree node ID, and all the other pieces of information will be stored in a lookup structure under this ID. The other integers on the list are used to construct an ancestor path for the new node in the decision tree model, and the tree model is adapted accordingly.

The call stacks are built on the Prolog side by retrieving a call ID for each call of a parsing predicate and pushing this on top of the list representing the call stack. The difference to a call trace is that when a called predicate succeeds and the calling predicate is continued, the ID of the called predicate does not vanish from the stack. The call stack should perhaps rather be called a decision stack, since the only point at which items are taken out of the stack is when Prolog backtraces to a point where some of the predicate calls that occurred later were not yet executed. This explains why the decision tree only branches at decision points and not as soon as a predicate calls multiple other predicates.

6 Known issues

- On certain systems, there might be problems with the relative classpath the system uses to tell Jasper where it can find the Java classes. In such cases, it usually helps to change the code to absolute classpaths. In effect, this means that the user must change the definition of `get_JVM/2`, which in the distributed version looks like this:

  ```
  get_jvm(JVM) :-
  jasper_initialize([classpath('./bin')],JVM).
  ```

  If you change the classpath to point to the `bin` folder in your installation directory, this will usually help, so change it e.g. to the following:

  ```
  get_jvm(JVM) :-
  jasper_initialize([classpath('/home/jdoe/lc-parse/bin')],JVM).
  ```

- Sometimes, the decision tree display tends to flicker once the GUI has been fired up. This is due to a bug in Swing, and there is not much you can do about it except forcing the window to redraw, e.g. by temporarily dragging another window over the debugger window or by minimizing and maximizing it once.

References

M. A. Covington (1994). *Natural Language Processing for Prolog Programmers.* Prentice Hall, Upper Saddle River, New Jersey.

E. T. Irons (1961). 'A syntax directed compiler for ALGOL 60'. *Communications of the ACM* **4**(1):51–55.

P. Resnik (1992). 'Left-corner parsing and psychological plausibility'. In *Proceedings of the 14th conference on Computational linguistics*, pp. 191–197, Morristown, NJ, USA. Association for Computational Linguistics.

D. J. Rosenkrantz & P. M. Lewis (1970). 'Deterministic left corner parsing'. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory*, pp. 139–152, Los Alamitos, CA, USA. IEEE Computer Society.

SICStus (2007). *SICStus Prolog User's Manual, Release 3.12.8.* Intelligent Systems Laboratory, Swedish Institute of Computer Science.