LR conflict demonstrator

Project in Computer Science – EDAN70
January 23, 2015

Daniel Eliasson Christian Olsson
Dept. of Computer Science, Lund University, Sweden
ada10del@student.lu.se christian.olsson86@gmail.com

Abstract

It can be difficult to resolve conflicts in a LR grammar. This tool hopes to help alleviate this problem. The tool finds a string of symbols leading up to a conflict in order to give an example of something that is ambigous to the parser. We do this by building a graph from the output dumped by the CUP parser generator and doing a search from the start node of the graph to the conflict state, generating a string of symbols.

1. Introduction

When you are writing a parser for a LR grammar it can be difficult to fix conflicts that appear. Most parser generators only tell you which productions are in a conflict, what type of conflict that occurs, and some additional information that can be hard to understand. This information is not enough to easily tell what kind of input would be problematic for the parser and having such an example input would make the conflict easier to resolve. What we have created during this project is a tool that takes the output from the CUP parser generator and from this finds an example string that would cause a conflict when parsed by a LALR-parser [2]. The example is a string of symbols that drives the parser to the conflict state. The idea is that this will give the programmer more contextual information leading up to the conflict.

The first step in using our tool is to dump the LR-graph from CUP. Then our tool will parse the dump recreating the LR-graph. Then we find a string of symbols that drive the parser to the conflict-state. This becomes a common prefix of symbols that lead up to the conflict. Later all symbols that are non-terminals could be expanded to create a string of tokens. It's later possible to turn tokens back into strings. Putting together those strings creates an example input.

2. Background

2.1 LR-parsing

Using LR-parsing it's possible to parse more grammars than using LL-parsing. The reason for this is that LL(k)-parsing techniques must predict which production to use after only having seen the first k tokens of the right-hand side. LR(k)-parsing is able to defer that decision until it has seen the entire right hand side of a production and another k tokens following that [3, p. 55].

The crucial difference is that LR-parsers have a stack of symbols and both the contents of the stack and the lookahead is used to choose the production. The entire right hand side of a production is put on to the stack before LR-parsers decide to choose that production. This means that productions that have a unlimited common prefix are not a problem for LR-parsers unlike LL-parsers. Like LL-parsing techniques, LR-parsing also uses a table of actions that describe what the parser should do. However, in a LR-parser this table has a column for every symbol and a row for every state in the LR-parser.

The LR-parser has a number of states. Each state contains a number of LR items. Each LR item is a production that the LR parser may be parsing in at this point. So each LR item is a production augmented with a position marker on the right hand side. So it has the form $A \rightarrow \alpha_1\alpha_2\alpha_3 \ldots \bullet \beta_1\beta_2\beta_3 \ldots$. The symbols to the left of the position marker $\alpha_1\alpha_2\alpha_3 \ldots$ are on the top of the stack when the parser is in this state. When the LR-parser reads in a token from the input and puts it on the stack it transitions to another state. The states in a LR-parser and the transitions between the states create a graph called the LR-graph. The LR-graph is later used to fill the parsing table with actions [1].

All LR-conflicts arise when two actions are put in the same position in the parsing table. However, the parser can only take one action at a time so we have a conflict. The conflict is either a shift-reduce or reduce-reduce conflict. A shift-reduce conflict occurs when the parser can either shift a terminal α or reduce according to a production that can be followed by α. Choosing one or the other can have a desired effect. However, figuring out which one can be difficult, especially as the error messages from parser generators are difficult to parse for humans (hence the need for this tool.). Reduce-reduce conflicts occur when the parser can match two productions, both which can be followed by the same symbol α, so it doesn't know which one to choose. Most of these problems should not be resolved by choosing one production over the other but by changing the grammar [3, p. 68].

2.2 Some examples of LR-parser conflicts

There are a lot of ways that you can run into a LR conflict. One of the more famous one that you might run into is the dangling else problem where you have two nested if statements after which an else clause follows. Depending on how exactly the grammar is defined the else clause can belong to either of the two if statements. If you're not careful when writing the grammar you'll get a shift-reduce conflict. Most parser generators will break the tie by favoring shift over reduce. In this case that means that the else clause will belong to the closest if statement. This might be what you want (and this is the case in C and Java) but you should make this explicit, by changing the grammar or hinting to the parser generator to choose shift.

In listing 2 we have en example of a very simple grammar with a shift/reduce conflict. It is impossible for the parser to know if `Expr + Expr * Expr`, should be interpreted as `(Expr + Expr) * Expr` or `Expr + (Expr * Expr)`. If you want the parser to adhere to normal math precedence you will have to change your grammar to reflect that or add precedence declarations.

Listing 1: Dangling else snippet

```
if (a)
   if (b) foo();
   else bar();
```

Listing 2: Simple grammar with shift/reduce conflict

```
Expr ::= Expr + Expr
Expr ::= Expr * Expr
```

In example 3 we have an example of a LALR(1)-grammar that has a reduce/reduce conflict. This is an example of a conflict-free LR(1) grammar that has a reduce/reduce conflict when used with a LALR(1) parser generator. Choosing one production over the other to resolve this will limit the LALR parser so that it cannot parse the whole language that the grammar describes. The grammar can be rewritten to be a conflict-free LALR(1)-grammar and our tool will hopefully help find and fix this type of problem.

Listing 3: Reduce/reduce conflict.

```
S ::= a E c
  ::= a F d
  ::= b F c
  ::= b E d
E ::= e
F ::= e
```

2.3 CUP

For this project we are using the CUP parser generator[1] to create LR-graphs from CUP grammars. It is possible to change our program to use an LR-graph from another source but for this project we are using CUP.

We use a fairly simple grammar to parse the output from CUP and with our program we can output the graph in .dot format or do analysis on it. In order to find a conflict state we only need to look at each state individually. In each state we look through the LR-items to find conflicts.

In CUP it is possible to resolve some conflicts using precedence rules. The advantage of this is that it makes it possible to write an ambiguous grammar that feels more natural to read. This is exceptionally useful in a language with expressions that include many different operators. But the conflicts can still be found in the LR-graph. It is when the LR-graph is converted to a parsing table that the conflicts will be resolved according to the precedence rules. This means that our tools will sometimes find conflicts in the grammar that CUP already knows how to resolve.

When working with LR-graphs from CUP it is not rare to come across a couple of special non-terminals. First there is the `$START` non-terminal that represents the start production. But then there are a couple of more mysterious non-terminals named `NT$0`, `NT$1`, `NT$2`... that occur in seemingly arbitrary places. These non-terminals are synthesized when semantic actions appear in the middle of a production. For example the following production:

```
A ::= B {: foo(); :} C {: bar() :} D;
```

[1] http://www2.cs.tum.edu/projects/cup/

Is transformed into:

```
A     ::= B NT$0 C NT$1 D;
NT$0 ::= {: foo(); :};
NT$1 ::= {: bar() :};
```

3. Implementation

Our tool is implemented with the help of JastAdd and the parser generator Beaver. We use Beaver to create a parser that can parse the output from CUP's dump states functionality to generate the LR-graph. JastAdd is an aspect-oriented compiler construction system. We use JastAdd to generate classes for an abstract syntax tree to represent the LR-graph. It also allows us to use attribute grammars and aspect oriented programming to add functionality to the graph [5].

When there are conflicts in a grammar, CUP will generate a list of these conflicts and also a summary at the end. Our scanner uses two states that skips everything before and after the LALR-states. Our representation of the LR-graph is basically a list of states where each state has a list of transitions to other states and each transition has an associated symbol with it. In other words this is basically an adjacency list representation. Each state also has a list of LR-items that we use to find conflicts.

Almost all the code is written as aspects as opposed to pure Java. We have aspects that deal with doing breadth-first search (BFS), finding conflicts, and outputting the graph in `.dot` format.

To find a conflict state all we need to do is to check each state in turn for conflicts. To check a state for conflicts we build a map from each symbol and follow the position marker to a list of the items that have that symbol after the position marker. That way all the LR-items that may cause a conflict are grouped together. If for any symbol there is more than one associated LR-item, and at least one of those LR-items is a reduce item, then there is a conflict.

To find a prefix to the conflict we choose one of the LR-items involved in the conflict. In particular one that is a reduce item (where the position marker is at the end of the production). Then we trace backwards in the graph to find the set of states that we can originate from when parsing in this production keeping note that the lookahead of the items must also match. Finally we find using BFS the shortest route from the start node to any of the set of nodes we found. This gives us the path that we are looking for.

3.1 Example: prefix to dangling else in a LALR-graph

As an example have a look at figure 2. This a LALR-graph of a grammar with the dangling else problem. There is a conflict in state 5 between the two items in that state. The first has the symbol `ELSE` following the position marker. The second item has the position marker at the end and it has that same symbol in its follow set. This means that we have a shift-reduce conflict because we have two possible actions. We can either shift `ELSE` onto the stack (first item) or we can reduce the top of the stack into a `IfStmt` symbol.

To find out where we came from we first begin tracing backwards using the second item from state 5. What we want to do is to find the set of states where we are at the beginning of that production. To do this we maintain two sets: one set of all of the current nodes C and one set of all the previous nodes P. Note that we are searching backwards so the set of all previous nodes are the nodes we are going to look at next.

In the beginning C only contains state 5 and the P is empty. Looking at the item we see that the way we got to state 5 was to shift `Stmt` onto the stack. So we look at each state in C for transitions on the `Stmt` symbol from other nodes. We find state 4 and the next step is to see if the item exist in this state with the `ELSE` symbol in the follow set. We find that it is the first item in state 4 and so we

add state 4 to P. Now there are no more nodes left so we assign P to C and take another step backwards.

Now when $C = \{4\}$, we want to follow all `Expr` transitions backwards. We find that $P = \{3\}$ and continue wth the next round. With $C = \{3\}$ we have one more step backwards to take along `IF` transitions. Here we find tree states: 0, 4, and 6. When looking at state 0 we see that it has a production matching the one we are following but it does not have `ELSE` in its follow set. This means that going along the path $0 \to 3 \to 4 \to 5$ cannot cause the conflict as it can not be followed by `ELSE` that is the cause of the conflict. So state 0 is discarded. Next state 4 and 6 are both inspected and added to P.

We have now found the set we were looking for: $\{4, 6\}$. Now we use BFS to find a path from state 0 to any of these nodes and find the path $0 \to 3 \to 4$ and to that path we append the path that we seached backwards along $4 \to 3 \to 4 \to 5$. This gives us the resulting path: $0 \to 3 \to 4 \to 3 \to 4 \to 5$ and picking out the symbols along the transitions we get the prefix to the conflict: `IF Expr IF Expr Stmt`. Add in the position marker and the conflict symbol and we get: `IF Expr IF Expr Stmt (*) ELSE`.

This may seem like an over complicated way of doing this. Our initial approch was to run BFS from the start state to the conflict. It appears that could work when using a LR-graph. But not when using an LALR-graph like we get from CUP. The path we want to find in the LALR-graph in figure 2 has a loop so we cannot find it using only BFS.

4. Evaluation of Our Tool

In order to evaluate our tool we used a few different metrics to look at different characteristics. Unfortunately we are not aware of any similar tools to compare against but we hope the numbers will be interesting nonetheless.

4.1 Lines of Code

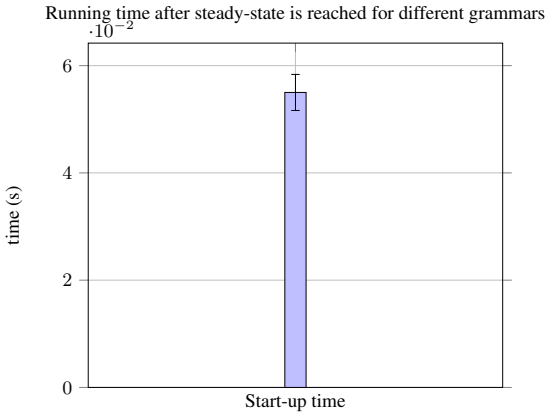
To evaluate how much code we wrote we can divide the project into two parts. The first part deals with parsing the LR-graph from CUP. For this part we wrote a combined 91 lines of AST specification, scanner rules, and AST production rules. The second part deals with analyzing the LR-graph. For this part we wrote 702 lines of code according to CLOC[2].

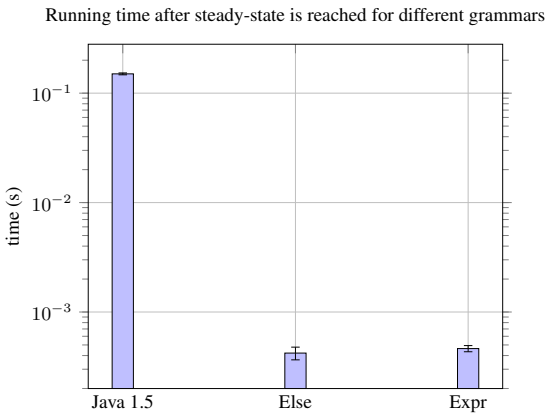
4.2 Running Time

We also measured the running time of our tool in a few different ways. In order to evaluate our tool we created a couple of different grammars. We also downloaded a CUP implementation of the Java 1.5 grammar[3]. The Java 1.5 grammar was changed to introduce a conflict (dangling else) for evaluation purposes. Some statistics about the LR-graphs created from the different grammars we used can be found in table 1. In order to measure the running time of the program we used the guidelines outlined in the OOPSLA[4] article. First we measured the start up performance: the time it takes for the JVM to start up and load the required libraries. We then measured the steady-state performance by making a loader class that ran the main program multiple times in the same JVM invocation until the coefficient of variation was less than 0.01 and then reported the mean running time for the last 10 iterations. With both these ways we ran the tests 30 times and computed a 95 % confidence interval. The tests where run on a laptop with an i7-4700MQ Intel processor using a SSD as a storage device.

[2] http://cloc.sourceforge.net/

[3] http://people.csail.mit.edu/jhbrown/Javagrammar/





As we can the running time time is quite acceptable even for a large grammar lika Java.

4.3 Time Complexity

It takes $\mathcal{O}(n) = \mathcal{O}(|V| + |E| + |L|)$ time to parse the LR-graph from the CUP output where $|V|$ is the number of states, $|E|$ is the number of transitions, and $|L|$ is the number of LR-items. Because $|V| < |E|$ in a LR-graph and $|L|$ can be assumed to be less than $k|V|$ where k is some constant, both time complexity and space complexity are $\mathcal{O}(|E|)$. Because the whole graph needs to be saved the space complexity is obviously the same. For an LR-graph (V, E) the time complexity for doing a BFS search to find the first state with an conflict is $\mathcal{O}(|E|)$ and the space complexity is $\mathcal{O}(|V|)$. The time complexity for finding a conflict in a state with K LR-items and with lookahead tokens of length at most L is $\mathcal{O}(|K| \cdot L)$. The number of lookahead tokens can't be more than the total number of terminals in the grammar. Because $|K|$ and L are both small we can ignore this cost and call the conflict finder $\mathcal{O}(1)$. This means that both the time and space complexity for the entire tool are $\mathcal{O}(|E|)$.

Testfile	$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$
Else	14	21	1.5
Java15	1028	11473	11.2
Expr	18	33	1.8

Table 1: Table showing the sparseness of the graphs

4.4 User Friendliness

Our tool is a fairly easy to understand command line tool with helpful error messages and a decent enough readme-file. It does

of course require some general knowledge about command line interfaces and the tool assumes you know how to generate dump files from CUP. The point of the tool is to give helpful knowledge about conflicts so let us compare our tool's output to how Beaver and CUP report conflicts.

Listing 4: Beaver example

```
parser.beaver: Warning: Resolved Shift-Reduce conflict
     by selecting (ELSE: SHIFT; goto 17) over (ELSE:
     REDUCE stmt = IF expr.e stmts.s) using precedence.
```

Listing 5: CUP example

```
Warning : *** Shift/Reduce conflict found in state #383
  between if_then ::= IF Expr Stmt (*)
  and     if_then_else ::= IF Expr Stmt (*) ELSE Stmt
  under symbol ELSE
  Resolved in favor of shifting.
```

Listing 6: Our tool

```
State 6 has a conflict. A prefix is:
IF Expr IF Expr Stmt ELSE
```

In this simple example grammar our tool is quite helpful but the output becomes a bit unwieldy for larger grammars.

5. Related Work

The closest we could find is the ANTLRWorks tool[4] that visualizes ambiguities in an interesting way (see appendix 3). ANTLRWorks is for LL-grammars so it has different conflicts and wouldn't work with Beaver or CUP. We are not aware of any related tools for LR or LALR grammars.

6. Future Work

During this project we realized that some things were not as simple as we thought, there were some things that we didn't have time for, and some things that were simply outside the scope of this project.

6.1 Generating a String of Terminals

When we first started we envisioned us first getting a string of symbols and from that generate a string of terminals. During the project we started questioning the use of doing this. We argued that `IF Expr IF Expr ELSE` was easier to understand than trying to expand the Expr into more concrete terms. Coming up with a concrete example would also require us to look at the scanner implementation.

After using our tool on the big Java 1.5 grammar however we realized that a lot of the non-terminals could be "expanded" into nothing. This would actually make the generated example easier to understand. Finding out which non-terminals are nullable in this way could produce examples which are a lot shorter as well.

6.2 Generating Parse-Trees

Generating the different parse trees would have been a helpful feature. That way you can easily see how the different interpretations of the LR-graph affects the outcome. Another thing related to this is that we could extract a smaller grammar that still contains the conflict. That grammar would contain only the parts needed for the parse tree to visualize the conflict.

[4] http://www.antlr3.org/works/

6.3 Handle Multiple Conflicts

While our program can find and print out which states have conflicts, it cannot print a common prefix for multiple conflicts. This one is a bit tricky since we are not sure exactly how the conflicts interact if there are multiple conflicts in the LR-graph.

7. Conclusions

During this project we have implemented a tool that can find a common prefix for a LR-conflict. It does this by finding a path to the closest conflict and adding the symbols associated with the transitions to the prefix string. A large problem we encountered during the project was the difference between LR- and LALR-graphs. In order to solve this we had to use a workaround which you can read about in section 3.

There were some things that unfortunately was outside the scope of this project which you can read about in Future Work. An unfortunate problem we did not manage to solve was that the prefix string becomes very long for large grammars. The problem could be alleviated if we could get rid of nullable terminals but we felt that that was outside the scope of this project.

We implemented a fast and efficient tool that we hope is helpful for anybody who writes parsers using CUP.

Acknowledgments

We would like to thank Jesper Öqvist for his helpful suggestions and for pushing us to start writing the report early.

References

[1] On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. ISSN 0019-9958. .

[2] T. Anderson, J. Eve, and J. Horning. Efficient lr (1) parsers. *Acta Informatica*, 2(1):12–39. ISSN 00015903.

[3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[4] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007. ISSN 0362-1340.

[5] G. Hedin and E. Magnusson. Jastadd-an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58.

A. Appendix

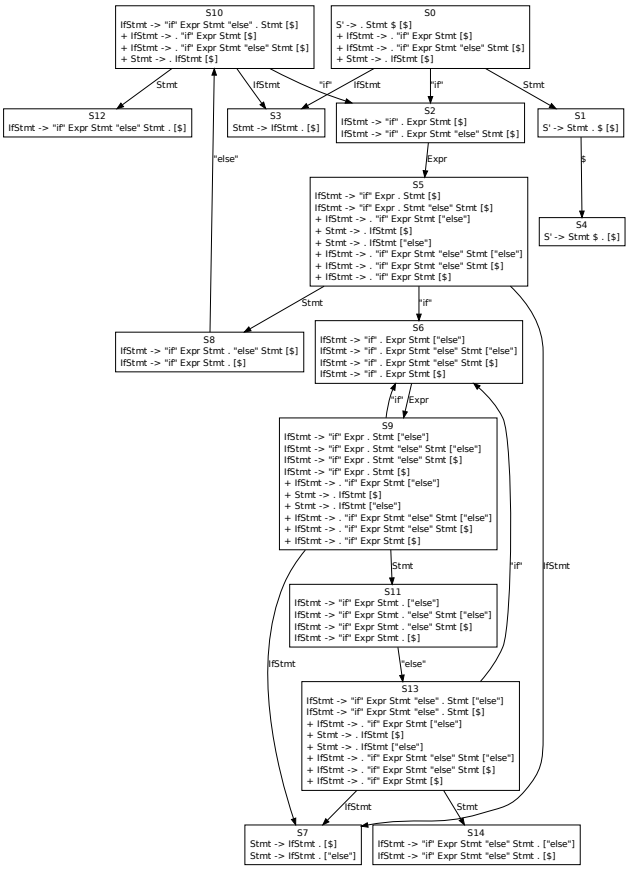


Figure 1: LR-graph for example grammar with dangling else. Generated by Jesper Öqvist's LR-parser.

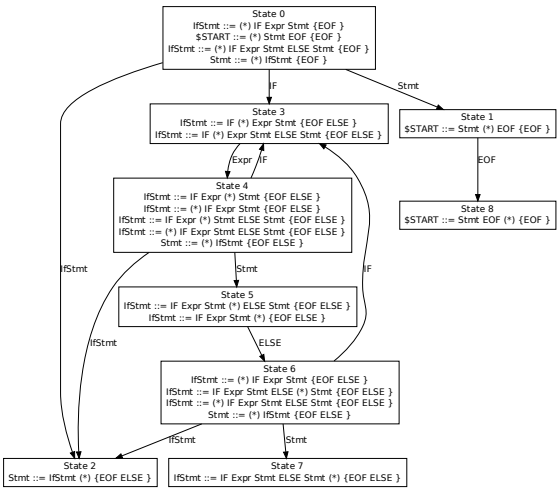


Figure 2: LALR-graph for example grammar with dangling else. Generated by our tool.

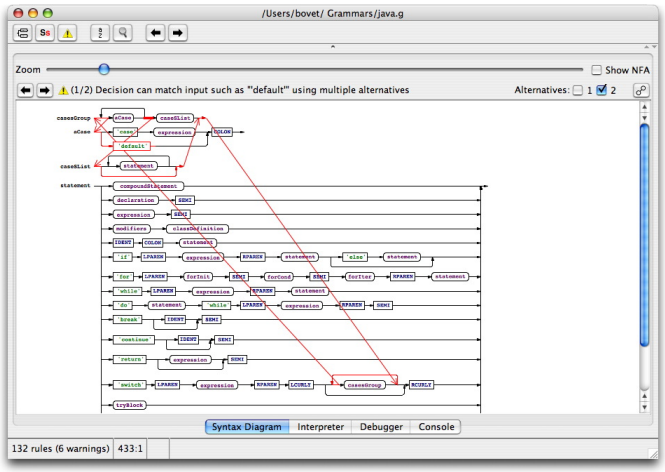


Figure 3: ANTLRWorks visualization of an ambigous path[5]

[5] http://www.antlr3.org/works/screenshots/ambiguouspath.jpg