A Methodology for Removing LALR(k) Conflicts

Leonardo Teixeira Passos, Mariza A. S. Bigonha, Roberto S. Bigonha

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
CEP: 31270-010 – Belo Horizonte – MG – Brazil

{leonardo, mariza, bigonha}@dcc.ufmg.br

Abstract. *Despite all the advances brought by LALR parsing method by DeRemer in the late 60's, conflicts reported by LALR parser generators are still removed in an old fashion and primitive manner, based on analysis of a huge amount of textual and low-level data stored on a single log file. For the purpose of minimizing the effort and time consumed in LALR conflict removal, which is definitely a laborious task, a methodology is proposed, along with the set of operations necessary to its realization. We also present a tool and the ideas behind it to support the methodology, plus its plugin facility, which permits the interpretation of virtually any syntax specification, regardless of the specification language used.*

Resumo. *Apesar de todos os avanços obtidos pelo método de análise sintática LALR de DeRemer no final da década de 60, conflitos reportados por geradores de analisadores sintáticos LALR ainda são removidos de forma primitiva, baseada na análise de uma grande quantidade de dados de baixo nível, disponibilizados em um único arquivo de* log. *Com o propósito de minimizar o esforço e o tempo gasto na remoção de conflitos, que é definitivamente uma tarefa laboriosa, uma metodologia é proposta, juntamente com as operações necessárias à sua realização. Além disto, apresenta-se uma ferramenta e as idéias utilizadas em sua criação no suporte à metodologia, acrescido da descrição da facilidade oferecida pelo mecanismo de* plugins*, que permite virtualmente a interpretação de qualquer especificação sintática, sem a preocupação da linguagem de especificação utilizada.*

1. Introduction

The great advantage of working with LALR(k) grammars is the fact that they can be used by *parser generators* to automatically produce fully operational and efficient parsers, encoded in languages like C, C++, Java, Haskell, etc. Examples of LALR parser generators are YACC [Johnson 1979], CUP [CUP 2007], Frown [Frown 2007], among others. However, the specification of a grammar that is indeed LALR(k) is not a trivial task, specially when k is limited to one, which is often the case. This happens due to the recurrent existence of *conflicts*, i.e., non-deterministic points in the parser. It is quite common in a typical programming language grammar being designed to find hundreds, if not more than a thousand conflicts. To illustrate that, when implementing the LALR(1) parser for the Notus [Tirelo and Bigonha 2006] language, whose grammar has 236 productions, 575 conflicts were reported by the parser generator.

There exists many approaches to remove conflicts. Those based on ad hoc solutions, such as precedence and associativity settings, are not considered by the methodology proposed herein. We favor the method based on rewriting some rules of the grammar, without changing the defined language.

A usual way to remove conflicts is to analyse the output file created by the parser generator. This output consists of a considerable amount of textual data, from the numerical code associated to grammar symbols to the grammar and the LALR automaton itself. Using the Notus language as an example, the Bison parser generator (the GNU version of YACC) dumps a 54 Kb file, containing 6244 words and 2257 lines. The big amount of data and the fact that none of it is interrelated – hyperlinks are not possible in text files, make it very difficult to browse. The level of abstraction in these log files is also a problem, since non experts in LALR parsing may not interpret them accordingly. When facing these difficulties, these users often migrate to LL parser generators. Despite their simplified theory, this approach is not a real advantage, since LL languages are a proper subset of the LALR ones. Even for experts users, removing conflicts in such harsh environment causes a decrease of productivity. To face this scenario, in this paper we present a methodology for removing conflicts in non LALR(k) grammars. This methodology consists of a set of steps whose intention is to capture the natural way the compiler designer acts when handling conflicts: (i) understand the cause of the conflict; (ii) classify it according to known conflict categories; (iii) rewrite some rules of the grammar to remove the conflict; (iv) resubmit the specification to make sure the conflict has been eliminated. Each of these steps comprises a set of operations that must be supported. The realization of these operations are discussed when presenting SAIDE [1], a supporting tool for the proposed methodology.

This article is organized as follows: Section 2 gives the necessary background to understand the formulations used in later sections; Section 3 discusses conflicts in LR and LALR parsing; Section 4 presents the proposed methodology; Section 5 presents SAIDE, the mentioned tool to support the methodology, and Section 6 concludes this article.

2. Background

Before we present the methodology itself, it is necessary to establish some formal concepts, conventions, definitions and theorems. Most of the subject defined here is merely a reproduction or sometimes a slight variation of what is described in [Charles 1991], [DeRemer and Pennello 1982], [Aho and Ullman 1972] and [Kristensen and Madsen 1981]. It is assumed that the reader is familiar with LR and LALR parsing.

A context free grammar (CFG) is given by $G = (N, \Sigma, P, S)$. N is a finite set of nonterminals, Σ the finite set of terminals, P the set of rules in G and finally $S \in N$ is the start symbol. $V = N \cup \Sigma$ is said to be the vocabulary of G. When not mentioned the opposite, a given grammar is considered to be in its augmented form, given by (N', Σ', P', S'), where $N' = \{S'\} \cup N$, $\Sigma' = \{\$\} \cup \Sigma$, $P' = \{S' \to S\$\} \cup P$, considering that $S' \notin N$ and $\$ \notin \Sigma$.

The following conventions are adopted: lower case greek letters (α, β, ...) define strings in V^*; lower case roman letters from the beginning of the alphabet (a, b, ...) and

[1] Correct pronunciation: /saɪd/

t, bold strings and operator characters ($+, -, =, .$, etc) represent symbols in Σ, whereas letters from the end of the alphabet (except for t) denote elements in Σ^*; upper case letters from the beginning of the alphabet (A, B, ...) and italic strings represent nonterminals in N, while those near the end (X, Y, ...) denote symbols in V. The empty string is given by λ and the EOF marker by \$. The length of a string γ is denoted as $|\gamma|$. The symbol Ω stands for the "undefined constant".

An LR(k) automaton LRA_k is defined as a tuple $(M_k, V, P, IS, GOTO_k, RED_k)$, where M_k is the finite set of states, V and P are as in G, IS is the initial state, $GOTO_k : M_k \times V^* \rightarrow M_k$ is the transition function and $RED_k : M_k \times \Sigma_k^* \rightarrow \mathcal{P}(P)$ is the reduction function, where $\Sigma_k^* = \{w \mid w \in \Sigma^* \wedge 0 \leq |w| \leq k\}$.

A state, either a LR or LALR one, is a group of items. An item is an element in $N \times V^* \times V^*$ and denoted as $A \rightarrow \alpha \bullet \beta$.

The usual way to build the LALR(k) automaton is to calculate the LRA_0 automaton first. For such, let the components of LRA_0 be defined.

The set of states is generated by the following equation:

$$\begin{aligned} M_0 \;=\; & \{F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\$\}))\} \cup \\ & \{F^{-1}(CLOSURE(F(q))) \mid q \in SUCC(p) \;\wedge\; p \in M_0\} \end{aligned}$$

where F is a bijective function that maps a state to a set of items (excluded the empty set) and

$$\begin{aligned} CLOSURE(is) &= is \cup \{B \rightarrow \bullet\beta \mid A \rightarrow \alpha \bullet B\omega \in is \;\wedge\; B \rightarrow \beta \in P\} \\ SUCC(p) &= \{F^{-1}(ADVANCE(p, X)) \mid X \in V\} \\ ADVANCE(p, X) &= \{A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X\beta \in F(p)\} \end{aligned}$$

The initial state (IS) is obtained by $F^{-1}(CLOSURE(\{S' \rightarrow \bullet S\$\}))$. $RED_0(q, w)$ is stated as

$$RED_0(q, w) = \{A \rightarrow \gamma \mid A \rightarrow \gamma\bullet \in F(q)\}$$

$GOTO_k, \forall k \geq 0$, can be defined as:

$$\begin{aligned} GOTO_k(p, \lambda) &= p \\ GOTO_k(p, X) &= F^{-1}(CLOSURE(ADVANCE(p, X))) \\ GOTO_k(p, X\alpha) &= GOTO_k(GOTO_k(p, X), \alpha), \forall\alpha \neq \lambda \end{aligned}$$

From this point, when mentioning a state p, it will be known from the context whether it refers to the number or to the set of items of the state.

The LALR(k) automaton, shortly $LALRA_k$, is a tuple $(M_0, V, P, IS, GOTO_k, RED_k)$, where except for RED_k, all components are as in LRA_0. Before considering RED_k, it is necessary to model a function to capture all predecessor states for a given state q, under a sentential form α. Let $PRED$ be such function:

$$PRED(q, \alpha) = \{p \mid GOTO_k(p, \alpha) = q\}$$

Then,

$$RED_k(q, w) = \{A \rightarrow \gamma \mid w \in LA_k(q, A \rightarrow \gamma\bullet)\}$$

where LA_k is the set of lookahead strings of length not greater than k that may follow a processed right hand side of a rule. It is given by

$$LA_k(q, A \rightarrow \gamma) = \{w \in FIRST_k(z) \mid S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A z \ \wedge \ \alpha\gamma \ access \ q\}$$

where

$$FIRST_k(\alpha) = \{x \mid (\alpha \overset{*}{\underset{lm}{\Rightarrow}} x\beta \ \wedge \ |x| = k) \ \vee \ (\alpha \overset{*}{\Rightarrow} x \ \wedge \ |x| < k)\}$$

and $\alpha\gamma$ *access* q iff $PRED(q, \alpha\gamma) \neq \emptyset$.

For $k = 1$, DeRemer and Pennello proposed an algorithm to calculate the lookaheads in LA_1 [DeRemer and Pennello 1982] and it still remains as the most efficient one [Charles 1991]. They define the computation of LA_1 in terms of $FOLLOW_1$: $(M_0 \times N \times M_0) \rightarrow \mathcal{P}(\Sigma)$. The domain $(M_0 \times N \times M_0)$ is said to be the set of nonterminal transitions. The first component is the source state, the second the transition symbol and the last one the destination state. For presentation issues, transitions will be written as pairs if destination states are irrelevant. $FOLLOW_1(p, A)$ models the lookahead tokens that follow A when ω becomes the current handle, as long as $A \rightarrow \omega \in P$. These tokens arise in three possible situations [DeRemer and Pennello 1982]:

a) $\exists\ C \rightarrow \theta \bullet B\eta \in p$, such that $p \in PRED(q, \beta)$, $B \rightarrow \beta A \gamma \in P$ and $\gamma \overset{*}{\Rightarrow} \lambda$. In this case, $FOLLOW_1(p, B) \subseteq FOLLOW_1(q, A)$. This situation is captured by a relation named *includes*: $(q, A)\ includes\ (p, B)$ iff the previous conditions are respected;
b) given a transition (p, A), every token that is directly read by a state q, as long as $GOTO_0(p, A) = q$, is in $LA_1(p, A)$. This is modeled by the direct read function:

$$DR(p, A) = \{t \in \Sigma \mid GOTO_0(q, t) \neq \Omega \ \wedge \ GOTO_0(p, A) = q\}$$

c) given (p, A), every token that is read after a sequence of nullable nonterminal transitions is in $LA_1(p, A)$. To model the sequence of nullable transitions the *reads* relation is introduced: $(p, A)\ reads\ (q, B)$ iff $GOTO_0(p, A) = q\ e\ B \overset{*}{\Rightarrow} \lambda$.

The function $READ_1(p, A)$ comprises situations (b) and (c):

$$READ_1(p, A) = DR(p, A) \cup \bigcup \{READ_1(q, B) \mid (p, A)\ reads\ (q, B)\}$$

From this and (a), $FOLLOW_1$ is written as:

$$FOLLOW_1(p, A) = READ_1(p, A) \cup \bigcup \{FOLLOW_1(q, B) \mid (p, A)\ includes\ (q, B)\}$$

Finally,

$$LA_1(q, A \rightarrow \omega) = \bigcup \{FOLLOW_1(p, A) \mid p \in PRED(q, \omega)\} \quad (1)$$

3. Conflicts in non LALR(k) grammars

Conflicts arise in grammars when, for a state q in the LALR(k) automaton and a lookahead string $w \in \Sigma^*$, such that $|w| \leq k$, at least one condition is satisfied:

a) $|RED_k(q, w)| \geq 2$: reduce/reduce conflict;
b) $|RED_k(q, w)| \geq 1 \land \exists A \rightarrow \alpha \bullet \beta \in q \land w \in FIRST_k(\beta)$: shift/reduce conflict.

If one of these conditions is true, q is said to be an inconsistent state. A grammar is LALR(k) if its correspondent LALR(k) automaton has no inconsistent states.

A conflict is caused either by ambiguity or lack of right context, resulting in four possible situations. Ambiguity conflicts are the class of conflicts caused by the use of grammar rules that result in at least two different parsing trees for a certain string. These conflicts cannot be solved by increasing the value of k; in fact there isn't a k (or $k = \infty$) such that the grammar is LALR(k). Some of these conflicts are solved by rewriting some grammar rules in order to make it LALR(k), according to the k used by the parser generator (situation (i)). As an example, consider the *dangling-else* conflict. It is well known that its syntax can be expressed by a non ambiguous LALR(1) set of rules, although is more probable that one will first write an ambiguous specification. Some ambiguity conflicts, on the other hand, simply cannot be removed from the grammar without altering the language in question (situation (ii)). These conflicts are due to the existence of inherently ambiguous syntax constructions. An example would be a set of rules to describe $\{a^m b^n c^k \mid m = n \lor n = k\}$.

The next class of conflicts are those that are caused by the lack of right context when no ambiguity is involved. These conflicts occur due to an insufficient quantity of lookaheads. A direct solution is to increase the value of k (situation (iii)). To illustrate this, consider the grammar fragment presented in Figure 1:

declaration	$\rightarrow$	*visibility exportable-declaration*
	$\mid$	*non-exportable-declaration*
non-exportable-declaration	$\rightarrow$	*function-definition*
visibility	$\rightarrow$	**public** $\mid$ **private** $\mid \lambda$
exportable-declaration	$\rightarrow$	*syntatic-domain*
syntatic-domain	$\rightarrow$	**domain-id** $=$ *domain-exp*
function-definition	$\rightarrow$	*temp4*
temp4	$\rightarrow$	*temp5* **id**
temp5	$\rightarrow$	**domain-id .**

Figure 1. Notus grammar fragment.

An LALR(1) parser generator would flag a shift/reduce conflict between items

temp5	$\rightarrow$	$\bullet$**domain-id .**
visibility	$\rightarrow$	$\lambda\bullet$

in a state q, for **domain-id** $\in LA_1(q, \textit{visibility} \rightarrow \lambda)$. However, the grammar is LALR(2), because the tokens after **domain-id** are either the equals sign ($=$), from *syntatic-domain* $\rightarrow$ **domain-id** $=$ *domain-exp*, or dot (**.**), from *temp5* $\rightarrow$ **domain-id ..**

However, even when no ambiguities are involved, there might be cases in which an infinite amount of lookahead is required (situation (iv)). In these cases, the solution to be tried is to rewrite some rules of the grammar without changing the language. Consider, for instance, the regular language $L = (b^+a) \cup (b^+b)$. A possible grammar for L is

$$
\begin{array}{lcl}
S & \rightarrow & A \mid B \\
A & \rightarrow & B_1\ \mathbf{a} \\
B & \rightarrow & B_2\ \mathbf{b} \\
B_1 & \rightarrow & B_1\ \mathbf{b} \mid \mathbf{b} \\
B_2 & \rightarrow & B_2\ \mathbf{b} \mid \mathbf{b}
\end{array}
$$

From the given productions, it is not possible to find a k for which the given grammar is conflict free. The reason is that $B_2 \rightarrow \mathbf{b}\bullet$ can be followed by an indefinite number of **b**'s when a conflict involving the item $B_2 \rightarrow \mathbf{b}\bullet$ is reported. The only possible solution for this example is to rewrite the grammar. For this simple example, such rewrite definitely exists, because L is a regular language. Nevertheless, it should be pointed out that this kind of solution is not always possible.

The mentioned four situations exhaust all possibilities of causes of conflicts in LALR(k) parser construction. These situations of conflicts are also applicable to LR(k) parser generation. One type of reduce/reduce conflict is, however, *LALR specific*. It arises when calculating LA_k for reduction items in states in M_0. Such calculation can be seen as generating the LRA_1 automaton and merging states with the same item set; lookaheads of reduction items in the new state are given by the union of the lookaheads in each reduction item in each merged state. When performing the merge, reduce/reduce conflicts, not present in the LR(1) automaton, can emerge. Specific LALR reduce/reduce conflicts occur if the items involved in the conflict do not share the same left context, i.e., a sentential form obtained by concatenating each entry symbol of the states in the path from IS to q, the inconsistent state. As a consequence, these conflicts do not represent ambiguity, but do not imply in the existence of a k.

4. The proposed methodology

The proposed methodology consists of four phases performed in an iteration while conflicts continue to be reported by the parser generator. These phases are: (i) understanding; (ii) classification; (iii) conflict removal and (iv) testing.

4.1. Understanding

To overcome the difficulty in analysing the data recorded in the log file dumped by the parser generator, this phase presents the same data available in the log file, but divided in proper parts, interrelated as hyperlinks. For example, when observing a state in the LALR(k) automaton, the user is able to directly visit the destination states given by the transitions in the currently state under visualization. The opposite operation should be possible as well, i.e., from a current state, grab all predecessor states. A modularized linked visualization of this data provides a better and faster browsing.

One drawback in visualizing this content is due to the low abstraction level it provides. While desired by expert users, this situation is not acceptable nor suitable for

analysis by non proficient users in LALR(k) parser construction. Therefore, one important characteristic of this phase is to provide high level data in order to understand the cause of the conflict. Derivation trees do meet this requisite, putting the user in a more comfortable position, as they approximate him/her to the real object of study - the syntax of the language, while reducing the amount of LALR parsing knowledge one must have in order to remove conflicts.

4.2. Classification

This phase aims to find one of the four situations described in Section 3 that gave rise to a conflict. Before removing it, a strategy must be planned. To understand the cause of the conflict is the first step for this, but the knowledge of the conflict's category adds much more confidence, as we strongly believe that a strategy used in removing a past conflict can be applied many times to other conflicts in the same category.

4.3. Conflict removal

Conflicts due to situation (iii) can be automatically removed. In the case of situation (i) the user is assisted with examples of solutions for known cases that match the current conflict. The removal, however, is performed manually. The methodology does not define operations for conflicts in situations (ii) and (iv).

4.4. Testing

The last step in conflict removal is testing. This should only be made in the case of a manual removal performed in the previous phase. To test, the user resubmits the grammar to the parser generator. As a result, it lists all conflicts found, plus the total amount of conflicts. The user checks this list, browsing it to make sure the conflict has indeed been eliminated.

5. SAIDE

SAIDE (*Syntax Analyser Integrated Development Environment*) is a tool, currently under construction, that aims to support the proposed methodology when it is applied to non LALR(1) grammars. Its main window is shown in Figure 2. The upper left corner frame contains the text editor with the opened syntax specification. This editor supports syntax highlighting and other common operations, such as searching, line and column positioning, undo, redo, cut, copy, paste, etc. The left bottom frame is the compilation window, the place where messages from the compilation of the the syntax specification are printed. A possible message is the report of a conflict. In this case, there is a link to allow its debug. A debug trace for the dangling else conflict is shown in the window at the right bottom. Finally, the last window in this figure is the LALR(1) automaton. Note that whenever possible, data is always linked, as indicated by underlined strings.

5.1. Realizing the methodology

This section outlines the algorithms used to support each phase of the proposed methodology.

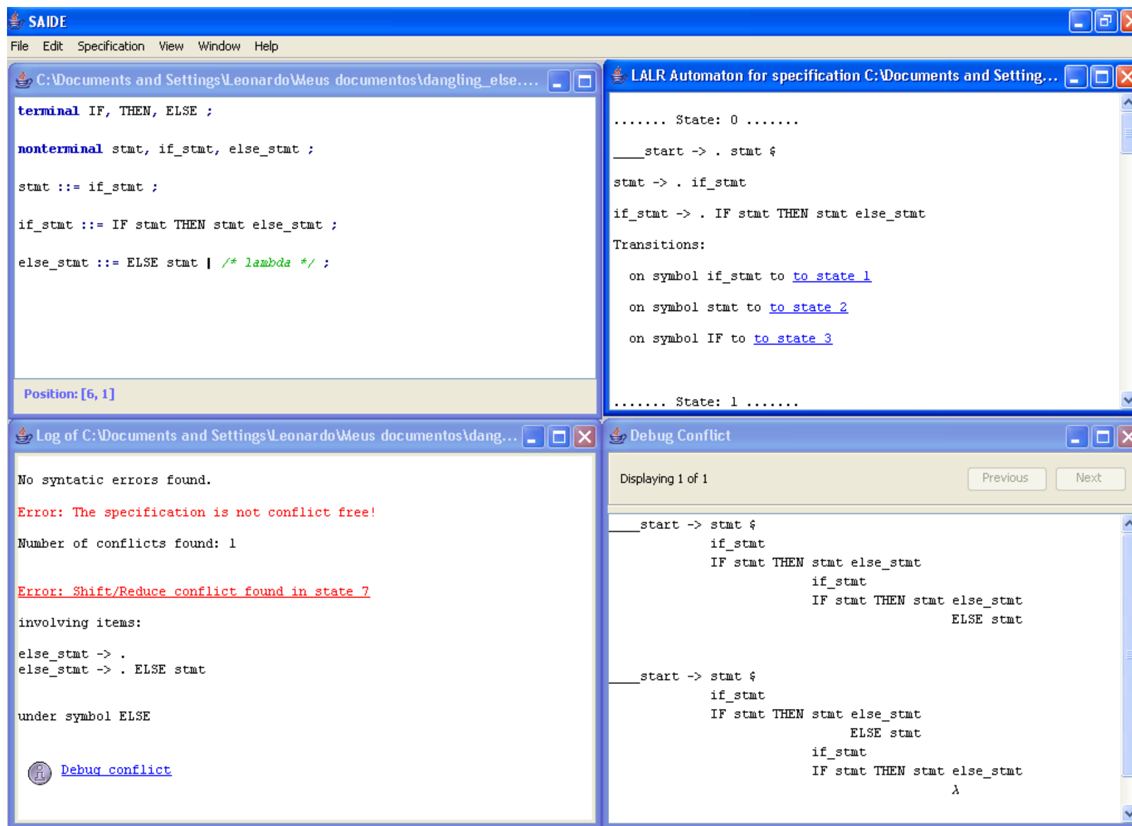

Figure 2. SAIDE's main window.

5.1.1. Understanding

To present the user with the LALR(1) automaton, first LRA_0 is obtained by the application of the $CLOSURE$ operation, as previously explained. The next step is to create the graphs corresponding to the *reads* and the *includes* relations. When there are two functions F and F' defined over a set of elements in X, and F is defined as $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}, \forall x, y \in X$, it follows that the nodes in a strongly connected component (SCC) found in the graph representing R have equal values for F [DeRemer and Pennello 1982]. Since $FOLLOW_1$ and $READ_1$ do match F's pattern, performing a search and identifying SCC's in the *reads* and *includes* graph permit the calculation of their value. The algorithm to efficiently perform this is presented in [DeRemer and Pennello 1982]. From the values in $READ_1$ and $FOLLOW_1$, the lookaheads of each reduction item are calculated using Equation 1, presented in Section 2.

To elucidate the cause of a conflict, SAIDE explains it in terms of derivation trees, as proposed by the methodology. Derivation trees are constructed for each reduction item. Their format is illustrated in Figure 3. The means to calculate parts (c), (b) and (a) are as follows [DeRemer and Pennello 1982]:

Part (c): given the item $A_{s-1} \rightarrow \alpha_s\bullet$ in an inconsistent state q, the traversal begins from the transitions in (q', A_{s-1}), where q' is in $PRED(q, \alpha_s)$, and then following some edges in the *includes* graph until a nonterminal transition (p, B) is found whose $READ_1$ set

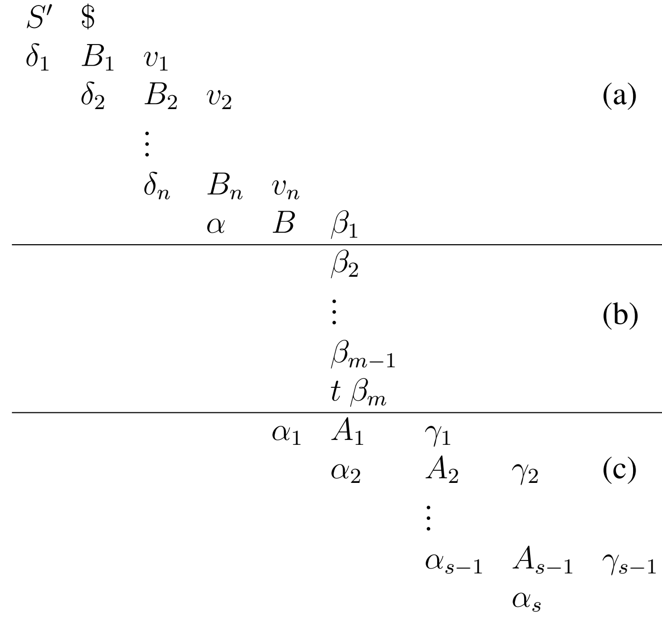

Figure 3. The format of a derivation tree.

contains the conflict symbol t. Every item $B_n \rightarrow \alpha \bullet B\beta_1 \in p$, such that $t \in FIRST_1(\beta_1)$, is considered a contributing one. For each of these, a debug should be printed separately.

To obtain the described path, a breadth-first search should be employed while traversing the *includes* graph. The production that induced an includes edge from (p, A) to (q, B) is rediscovered by following the automaton transitions from state q under the right parts of B productions.

Part (b): the next step is to get a derivation from β_1 until t appears as first token. To do this, the set

$$\begin{aligned} E \;=\; & \{B_n \rightarrow \alpha B \bullet \beta_1\} \\ & \cup \{A \rightarrow \delta X \bullet \eta \mid A \rightarrow \delta \bullet X\eta \in E \wedge X \stackrel{*}{\Rightarrow} \lambda\} \\ & \cup \{C \rightarrow \bullet\alpha \mid A \rightarrow \delta \bullet C\eta \in E \wedge C \rightarrow \alpha \in P\} \end{aligned} \tag{2}$$

is calculated. Each addition to E must be linked back to the items that generated it. Items of the form $C \rightarrow \bullet t\beta_m$, being t a conflict symbol, will be in E and are traced back to $B_n \rightarrow \alpha B \bullet \beta_1$ by following these links.

Part (a): calculate the derivation from S' that gave rise to $B_n \rightarrow \alpha \bullet B\beta_1$. First, it is necessary to find the shortest path from the start state to the contributing state, i.e., the state in which $B_n \rightarrow \alpha \bullet B\beta_1$ appeared. ξ is the sequence of transition symbols in this path. Then,

$$\begin{aligned} E' \;=\; & \{(S' \rightarrow \bullet S\$, 1)\} \\ & \cup \{(C \rightarrow \bullet\alpha, j) \mid (A \rightarrow \delta \bullet C\eta, j) \in E' \wedge C \rightarrow \alpha \in P)\} \\ & \cup \{(A \rightarrow \delta X \bullet \eta, j+1) \mid (A \rightarrow \delta \bullet X\eta, j) \in E' \wedge X = \xi_j \wedge j \leq |\xi|\} \end{aligned} \tag{3}$$

is calculated in a breadth first search, linking additions to E' back to pairs that generated

them. When $(B_n \rightarrow \alpha \bullet B\beta_1, |\xi|)$ appears, the computation stops. The derivation is given by the links created when elements were added.

To debug a reduce/reduce conflict, the sequence (c) $\rightarrow$ (b) $\rightarrow$ (a) is applied to each reduction item, and printed to the user. Sometimes, the left context formed by $\delta_1\delta_2...\delta_n\alpha\alpha_1...\alpha_s$ happens to be different from one reduction debug tree to the other. This indicates that the conflict in question is LALR specific, and is accordingly indicated by the tool.

When a conflict presents a shift, its corresponding tree is obtained by using an algorithm that implements Equation 3. The only difference is that ξ now corresponds to the symbols in $\delta_1\delta_2...\delta_n\alpha\alpha_1...\alpha_s$.

5.1.2. Classification

The classification aims to find the category to which the conflicts belongs. Each category represents one of the four situations explained in Section 3.

At this point, SAIDE first attempts to find a value of k, limited by a parameter value, say k_{max}, capable of giving enough right context to allow the removal of the conflict situation (iii). The value of k_{max} is read from a configuration file at SAIDE's start up and its default value is 3. Before presenting how the correct value of k is achieved, it follows a discussion of how a given lookahead w, such that $|w| \leq k$, is found.

Charles [Charles 1991] proposes Equation 4 to calculate the $FOLLOW_k$:

$$\begin{array}{lcl} FOLLOW_0(p, A) & = & \{\lambda\} \\ FOLLOW_k(p, A) & = & \quad READ_k(p, A) \\ & & \cup \bigcup\{FOLLOW_k(p', B) \mid (p, A) \; includes \; (p', B)\} \\ & & \cup \bigcup\{CONCAT(\{w\}, FOLLOW_{k-|w|}(q, B)) \mid \\ & & \qquad w \in SHORT_k(p, A), \\ & & \qquad B \rightarrow \alpha \bullet A\beta \in p, \\ & & \qquad q \in PRED(p, \alpha), \\ & & \qquad B \neq S\} \end{array} \tag{4}$$

where

$$\begin{array}{lcl} SHORT_k(p, A) & = & \{w \mid w \in FIRST_k(\beta), 0 < |w| < k, B \rightarrow \alpha \bullet A\beta \; \in p\} \\ READ_k(p, A) & = & \{w \mid w \in FIRST_k(\beta), |w| = k, B \rightarrow \alpha \bullet A\beta \in p\} \end{array}$$

and $CONCAT(M, N)$ is defined as $\{mn \mid m \in M \wedge n \in N\}$.

Charles presents an algorithm to calculate $READ_k$ and $SHORT_k$ based on the simulation of the steps performed by the LRA_0 automaton. His algorithm is directly based on Equation 5 [Charles 1991]:

$$
\begin{array}{rcl}
READ_{0*}(stack, X) & = & \{\lambda\} \\
READ_{k*}(stack, X) & = & \bigcup\{CONCAT(\{a\}, READ_{(k-1)*}(stack + [q], a) \mid \\
& & \quad a \in DR(TOP(stack), X) \\
& & \quad q = GOTO_0(TOP(stack), X)\} \\
& & \cup \bigcup\{READ_{k*}(stack + [q], Y) \mid \\
& & \quad (TOP(stack), X) \ reads \ (q, Y)\} \\
& & \cup \bigcup\{READ_{k*}(stack(1...SIZE(stack) - |\gamma|), C) \mid \\
& & \quad C \rightarrow \gamma \bullet X \in TOP(stack), \\
& & \quad |\gamma| + 1 < SIZE(stack)\}
\end{array}
\tag{5}
$$

$SHORT_k$ and $READ_k$ are then rewritten as:

$$
\begin{array}{rcl}
SHORT_k(p, A) & = & \{w \mid w \in READ_{k*}([p], A), 0 < |w| < k, B \rightarrow \alpha \bullet A\beta \ \in p\} \\
READ_k(p, A) & = & \{w \mid w \in READ_{k*}([p], A), |w| = k, B \rightarrow \alpha \bullet A\beta \in p\}
\end{array}
$$

Charles states that in the presence of cycles in the grammar, i.e., nonterminals that rightmost produce themselves and SCC's in the *reads* graph, his algorithm may not terminate. To guarantee termination, a verification of the non occurrence of these two conditions must always be performed. Later, the author discards Equation 4 as the bases of an algorithm to calculate $FOLLOW_k$. His main argument is that it does not match the format $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$.

However, Equation 5 can still be used to calculate the strings that are read from a given transition even when the conditions pointed by Charles are not true. This is achieved if one keeps track of every reached stack and the string read so far while simulating the LRA_0 steps. Kristensen and Madsen [Kristensen and Madsen 1981] argue that a tree can be used to store such data. A node in this tree is a pair of the form $(M_0 \times \Sigma^*)$ and maps to a unique configuration, i.e., a stack of states and the corresponding string read at the moment. The correspondence between a node n and a configuration is guaranteed in the following way: the states in the path from the root node of the tree to n forms a stack. The string obtained by such stack is the string stored in n. During the simulated parsing, the tree will be expanded with a node each time a transition is carried out. If an attempt to add a node that is already in the tree is performed, then circularity is detected, and thus cycles are controlled.

A straightforward algorithm using these ideas and Equations 4 and 5 can be constructed. Such algorithm would clearly terminate, as it would depend solely on the values of $READ_{k*}(p, A)$ and $FOLLOW_k(q, B)$, where (p, A) $includes$ (q, B). This fact is attested by:

i) if (p, A) and (q, B) belong to an SCC in the *includes* graph, then their lookaheads are equal. This is assured because Equation 4 matches the format $F(x) = F'(x) \cup \bigcup\{F(y) \mid xRy\}$, where X is the set of nonterminal transitions and $F'(p, A)$ is

given by

$$\begin{aligned} F'(p, A) \quad = \quad & READ_k(p, A) \\ & \cup \bigcup \{CONCAT(\{w\}, FOLLOW_{k-|w|}(q, B)) \mid \\ & \quad w \in SHORT_k(p, A), \\ & \quad B \rightarrow \alpha \bullet A\beta \in p, \\ & \quad q \in PRED(p, \alpha), \\ & \quad B \neq S\} \end{aligned}$$

This matching permits an easy control of cycles.

ii) $READ_{k*}$ is calculated using Equation 5. Storing each stack and the string read so far at each step in the calculation of $READ_{k*}$ permits the control of cycles, thus, preventing the algorithm to loop forever.

Furthermore, no restrictions are considered in the LRA_0 or any relation graph.

We are currently implementing an algorithm based on the discussed equations and cycle control scheme to generate the values in LA_k to determine the value of k necessary to solve a conflict. Its iteration starts with $k = 2$. If RED_k continues to report the conflict, a new iteration is performed, which increments the value of k. This continues until the conflict is "removed" or k becomes greater than k_{max}. The algorithm caches all calculated sets, since they might be used later for other conflicts.

If the classification fails to find a $k \leq k_{max}$ capable of removing the conflict and it is a non reduce/reduce conflict specific to LALR, the next attempt is ambiguity detection (situation (i)). It is known from the literature that this problem is undecidable. Therefore, a study to capture recurrent cases was performed and some patterns were noticed. A pattern consists of two sentential forms derivable from a nonterminal P. Expanding each sentential form will eventually lead to the ambiguity in study. These patterns were inferred from ambiguities found in the grammars of the programming languages Notus, Algol60 and Oberon2.

For each pattern, it must be asserted that $S' \stackrel{*}{\Rightarrow} \xi' P \xi''$, $\delta_i \stackrel{*}{\Rightarrow} \lambda$ and $P_i \stackrel{*}{\Rightarrow} P$. The symbols δ_i and P_i are used in the definition of the filters listed bellow:

Filter 1)
Pattern: $P \stackrel{*}{\Rightarrow} \delta_1 P_1 \delta_2 \alpha \delta_3 P_2 \delta_4$ and $P \stackrel{*}{\Rightarrow} \delta_6 \beta \delta_7 \delta_8$. This filter captures ambiguous constructions such as $E \rightarrow E + E \mid t$.

Filter 2) Pattern: $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3$ and $P \stackrel{*}{\Rightarrow} \delta_5 \alpha P_2 \delta_6 \beta \delta_7 P_3 \delta_8$. This filter identifies dangling-else's instances.

Filter 3) $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3$ and $P \stackrel{*}{\Rightarrow} \delta_5 P_2 \delta_6 \beta \delta_7$. This filter captures ambiguous constructions such as the rules *exp* → **let** *dcl* **in** *exp* **where** *exp* and *exp* → *exp* **where** *exp*.

Filter 4) $P \stackrel{*}{\Rightarrow} \delta_1 \alpha \delta_2 P_1 \delta_3 \beta \delta_4$ and $P \stackrel{*}{\Rightarrow} \delta_6 \alpha \delta_7 P_2 \delta_8 \beta \delta_9$. This filter captures alias between nonterminals.

We are managing to apply these filters on the derivation trees obtained by the first step of the methodology.

If a conflict is due to situation (ii) and (iv), SAIDE is unable to classify and assist the user.

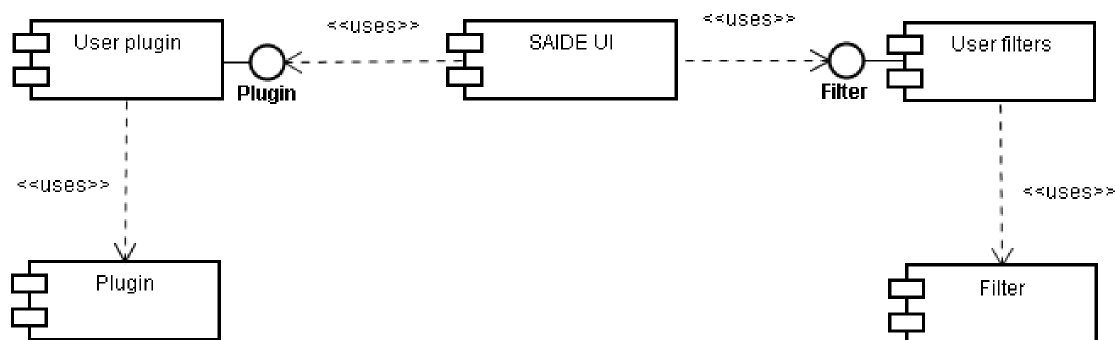

Figure 4. SAIDE's architecture illustrated as a component diagram in UML.

5.1.3. Conflict removal

Automatic conflict removal can only be accomplished if the parser is LALR(k) and $k \leq k_{max}$. There are two approaches for this problem.

The first one is to rewrite the grammar, starting from the productions involved in the conflict so that no reduction moves are performed until k tokens are read. [Mickunas et al. 1976] proposes a technique to transform LR(k) grammars into LR(1) correspondent ones, but it can deeply change the structure and the number of the rules in the grammar and is not generalized to LALR(k) grammars.

The other approach consists in the generation of an LALR(k) parser whose k varies. A conflict is removed if the parser generator, in this case SAIDE, can attest that nondeterminism is removed after examining k tokens ahead. The value of k in this case is local and is intended to solve only the given conflict.

5.1.4. Testing

To check if the conflict was been wiped out, SAIDE list all conflicts along with the total sum of conflicts found. The user browses this list and compares the current results with the ones previously presented.

5.2. Plugin facility

SAIDE's architecture, shown in Figure 4, permits its extensibility via plugins. A plugin instance must implement an interface with two methods responsible for returning `PluginParserFactory` and `HighlightLexerFactory` objects. `PluginParserFactory` is used by SAIDE to instantiate a parser capable of processing the specification file. The parsing result is an `Specification` instance used by the tool to generate data structures such as the LALR(1) automaton, *includes* and *reads* graphs, etc. and it appears throughout SAIDE's code. SAIDE's architecture permits the use of virtually any syntax specification language as long as there is a plugin implemented. In a similar way, the tool can be extended with filters in addition to the ones made available.

6. Conclusion

In this article, we presented the problem of conflict removal in non LALR(k) grammars. It was argued that this remains an arduous and time consuming task, considering that users continue to remove conflicts analysing extensive log files.

A methodology was proposed and the algorithms necessary to realize it were presented, showing the tool SAIDE.

The methodology is an important contribution to LALR parsing development and so is the outlined algorithm discussed in Section 5.1.2 for the calculation of lookahead strings of length not greater than k. Two important properties of such algorithm are: it is guaranteed to terminate, and no pre-conditions must be checked before running it.

At the present time, we are implementing the ambiguity detector, which will attempt to detect some previous cataloged ambiguity instances.

As future work, we intend to formulate an algorithm to automatically remove a subset of conflicts, either by redefinition of some rules of the grammar or generating an LALR parser whose k varies.

References

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference.

Charles, P. (1991). *A Pratical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis.

CUP (2007). Cup: Lalr parser generator in java. `http://www2.cs.tum.edu/projects/cup/`. Last access: 01/13/2007.

DeRemer, F. and Pennello, T. (1982). Efficient computation of lalr(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649.

Frown (2007). Frown - an lalr(k) parser generator for haskell. `http://www.informatik.uni-bonn.de/~ralf/frown/index.html`. Last access: 01/13/2007.

Johnson, S. (1979). Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston.

Kristensen, B. B. and Madsen, O. L. (1981). Methods for computing lalr(k) lookahead. *ACM Trans. Program. Lang. Syst.*, 3(1):60–82.

Mickunas, M. D., Lancaster, R. L., and Schneider, V. B. (1976). Transforming lr(k) grammars to lr(1), slr(1), and (1,1) bounded right-context grammars. *J. ACM*, 23(3):511–533.

Tirelo, F. and Bigonha, R. (2006). Notus. Technical report, Federal University of Minas Gerais - Department of Computer Science, Programming Languages Laboratory.