An LALR Parser Generator Supporting Conflict Resolution

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 advance brought by LALR parsing method by DeRemer in the late 60’s, conflicts continue to be removed in a non-productive way, by means of analysis of a huge amount of textual and low level data dumped by the parser generator tool. For the purpose of changing this scenario, we present a parser generator capable of automatically removing some types of conflicts, along with a supported methodology that guides the process of manual removal. We also discuss the internal algorithms and how the created parsers are compact in terms of memory usage.

Resumo. Apesar de todo o avanço obtido pelo método de análise sintática LALR criado por DeRemer no fim dos anos 60, conflitos ainda são removidos de forma não produtiva, pela análise de extensos arquivos de log criados por geradores de analisadores sintáticos. De forma a alterar este cenário, apresentamos um gerador de analisador sintático capaz de remover automaticamente certos tipos de conflitos, em conjunto com uma metodologia que guia o processo de remoção manual. Discutimos também os algoritmos internos da ferramenta e como os analisadores sintáticos produzidos são compactos em termos de utilização de memória.

1. Introduction

The usual way to build a bottom-up syntax analyser is by writing syntax specifications that are readable by parser generator tools, such as YACC, Bison, CUP, etc. These tools can only work with LALR(1) grammars, otherwise conflicts are reported, i.e., non-determinism points resulted from the grammar. The number of conflicts in a programming language can easily reach hundreds, if not more than a thousand conflicts. To illustrate how frequent conflicts are, the original grammars of Algol-60, Scheme and Notus[Tirelo and Bigonha 2006] programming languages result in 61, 78 and 575 conflicts, respectively. In the first two grammars, the average density is one conflict for each two productions; in the latter there are two per production.

To remove the conflicts in a syntax specification, one must inspect each conflict. Parser generators assist users in conflict removal by writing log files. These files consist of pure text data such as the list of conflicts and the LALR(1) automaton. Log files, however, tend to be too extensive for analysis. For the Notus programming language, Bison creates a log file of 54 Kb in size, having 6,244 words and 2,257 lines. Besides its size, log files contain a low level of abstraction in their content, requiring expertise in LALR inner working. All together, these factors contribute to a low productivity when building LALR parsers.
In order to change the current scenario, we present in this article an LALR parser
generator that automatically removes conflicts and supports a methodology to guide the
process in cases of manual removal. Such methodology is the result of new techniques that
extend the original work presented in [Passos et al. 2007]. In addition, we also discuss
how the proposed tool generates parsers with little memory requirements.

This article is organized as follows: Section 2 discusses the types of conflicts,
Section 3 presents the proposed parser generator, with its algorithms explained in Section
4. Section 5 presents some experimental results and Section 6 concludes the article.

2. Types of Conflicts
A conflict, either a shift/reduce or reduce/reduce, is reported by the parser generator by
two reasons: lack of right context and ambiguity.

Conflicts caused by lack of right context indicate that the number of inspected
lookaheads is not enough to decide which action to execute – shift or reduce within a set
of possible reductions. Part of these conflicts can be removed if the value of \( k \) is increased
accordingly. This results in an LALR\((k)\) parser, where \( k = \max\{k' \mid k' \text{ is the number of lookaheads needed to solve a given conflict among all conflicts reported by the parser generator and } k \text{ is not infinite}\}\). This solution, although correct, is unfeasible in
most cases. If we take, for example, the LALR(1) parsing table of Visual Basic .NET \(^1\)
and make it LALR(3), the result matrix would contain 1,899,085,824 entries. The other
part of this set of conflicts can only be removed by rewriting the grammar, assuming
the language in question is indeed LALR. Such cases result from an infinite amount of
lookaheads necessary to solve a given conflict.

Conflicts caused by ambiguity must be removed by rewriting the grammar. Again,
this can only be performed if the language in question is LALR. Some parser generators
deal with ambiguity by means of precedence and associativity or simply by performing a
shift in a shift/reduce conflict.

A complete discussion of all types of conflicts can be found in [Passos et al. 2007].

3. Proposed Tool
The proposed tool, named SAIDE\(^2\), is an integrated development environment with an
internal parser generator. Figure 1 shows the overall appearance of SAIDE’s graphical
interface. The text editor window is located in the upper left corner, loaded with a syntax
specification. After asking for the validation of the grammar, the user starts the main cycle
of the methodology supported by the tool. The main cycle is divided in two major steps:
automatic and manual conflict removal.

In the automatic removal, SAIDE tries to remove all conflicts without user inter-
vention. The non-removed conflicts are then listed to the user. This listing is performed
using a heuristic that sorts all conflicts considering the order in which they must be re-
moved. A conflict must be listed before those that appear as a consequence of the exist-
tence of the first. In order to calculate such removal priority, SAIDE needs to know the
whole set of conflicts. In Figure 1, the listing of conflicts is shown below the editor.

\(^1\)This test was performed using the grammar provided in http://www.devincook.com/goldparser/grammars/index.html.
\(^2\)SAIDE /sai/; Syntax Analyser Integrated Development Environment.
After the listing, the manual removal step starts. According to the methodology, to manually remove a conflict one must go through four phases: (i) understanding; (ii) classification; (iii) editing and (iv) testing.

In the understanding phase, the user tries to deduce the cause of the conflict using derivation trees. This has the advantage of manipulating a more intuitive and higher level structure compared to the low level data available in log files. Derivation trees are presented after the user clicks on the *Debug conflict* option, shown as a hyperlink below the conflict’s item set. For expert users, low level content is still available, as can be seen by the LALR automaton shown next to the editor window, in Figure 1.

In the classification phase the user defines the category in which the conflict belongs, i.e., determine whether a conflict is due to lack of right context or ambiguity. In the latter case, a catalog of some well known ambiguity constructions, along with their solutions, is available for consultancy and can be extended with user defined entries. At this phase, the user must define a strategy to rewrite the grammar so the given conflict can be removed. The identification of the conflict’s category adds confidence, as we expect that a strategy used in removing a past conflict can be applied many times to other conflicts in the same category.

Next, the user edits the grammar in order to apply the strategy defined in the last phase and submits the specification to be validated. The main cycle of the methodology is then restarted and continues until no conflicts are reported.

### 4. Automatic Conflict Resolution

Deviating from conflict resolution based on nondeterminism like Generalized LR Parsing [Tomita 1991], we address an automatic and deterministic resolution approach. In this section we give an overview of the technique proposed by Charles [Charles 1991], capable of removing some conflicts caused by lack of right context.

Charles suggests LALR($k$) generation as a mechanism to remove conflicts. To perform this while decreasing storage needs, he proposes the extension of the number
of inspected lookaheads only when necessary to uniquely define which parse action to execute. Thus, the parser inspects up to \( k \) tokens before choosing a parse action. The value of \( k \) is limited by a constant \( k_{\text{max}} \). This approach has the advantage of reducing the number of entries in the LALR parsing table when compared to complete LALR\((k)\) tables. These parsers will be referred as LALR\((k_v)\), with \( k_v \) denoting the variable length characteristic of \( k \).

To build the LALR\((k_v)\) parser, each entry \((p,a)\) in the action table with at least one conflict will become the start state of a deterministic finite automaton (DFA). From \( p \) following the conflict symbol \( a \), another state, represented as \( q \), is reached. For all tokens \( t_1, ..., t_n \) that may follow \( a \), the parsing actions in \((q,t_i)\) are properly set. If at this point, all \((q,t_i)\) contain cardinality equal to one, then all conflicts have been removed. Otherwise, if there is a least one entry \((q,t_i)\) whose cardinality is greater than one, another level of lookaheads is calculated. These consist of the tokens that may follow \( t_i \), given a conflictuous entry \((q,t_i)\). This process continues until the conflicts are removed or the depth of the DFA reaches \( k_{\text{max}} \). The described scheme is depicted in terms of an example in Figure 2. Suppose, that the original LALR parsing table entry \((q,a)\) has the following actions: \(\{S2, R5, R6\}\). In this entry, there are three conflicts: two shift/reduces (\(\{S2, R5\}, \{S2, R6\}\)) and a reduce/reduce (\(\{R5, R6\}\)). If \( k_{\text{max}} \) is taken as three, \( q \) is made the start state of the DFA. From state \( q \) following \( a \), the destination \( q_1 \) is reached. If \( b \) or \( c \) follow \( a \), then a unique parsing action is determined. At this point, the shift/reduce conflicts vanish. The entry, \((q_1,d)\) still reports \(\{R5, R6\}\), and the automaton is extended in another level of lookaheads. The tokens that can be found after \( d \) are \(\{b, c\}\). Extending \( d \) with these tokens uniquely determines each reduction. Note that although \( k_{\text{max}} \) was three, in \( q_1 \) only two lookaheads were used.

The action parsing table of an LALR\((k_v)\) parser is encoded as follows: each action that points to a DFA becomes a lookahead action – \(L_n\), where \( n \) is the first line available in the table. The rest of the DFA’s transition table is appended starting from the \( n \)-th line of the action parsing table. Note that this coding scheme preserves the original layout of the LALR\((1)\) action parsing table, since the columns are still indexed by a single token, in contrast with LALR\((k)\) encoding method. To illustrate the produced table, consider the following grammar:
This grammar is not LALR(1); it is LALR(2). The following action parsing table results from the LALR($k_v$) approach:

<table>
<thead>
<tr>
<th>Action</th>
<th>s &quot;→&quot; &quot;$&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R3</td>
</tr>
<tr>
<td>1</td>
<td>S3</td>
</tr>
<tr>
<td>2</td>
<td>ACC</td>
</tr>
<tr>
<td>3</td>
<td>S5</td>
</tr>
<tr>
<td>4</td>
<td>R2</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
</tr>
<tr>
<td>6</td>
<td>L8</td>
</tr>
<tr>
<td>7</td>
<td>R5</td>
</tr>
</tbody>
</table>

Lookaheads: 8 S7 R4 S7

An LALR($k_v$) parser works almost as an LALR($k$) parser. The main difference occurs when dealing with lookahead actions. Given an entry $Ln$ in position $(q, a)$ in the action table, the parser switches to the appropriate DFA table. By consulting the tokens that may follow $a$ in the input, the parser tries to find a path from the current DFA state that leads to a non-error parse action (different from blank). This possibly implies in making other lookahead actions, that differ from conventional shift actions in the sense that the tokens consulted in the input are not consumed in any way.

Charles’ algorithms to calculate the action tables for LALR($k_v$) parsers restrict the input grammars to the set of grammars that: (a) do not have circularity, i.e., given a nonterminal $A$, $A$ derives $A$ in one or more steps; (b) do not imply in the existence of strong connected components (SCCs) in the reads relation graph. The reads graph represents the relation [DeRemer and Pennello 1982]:

$$(p, A) \text{ reads } (q, B) \text{ iff } \text{GOTO}_0(p, A) = q e B \Rightarrow \lambda$$

where GOTO$_0$ is the transition function of the LALR automaton, defined over $(M_0 \times V) \rightarrow M_0$. The set $M_0$ contains all LR(0) states and $V$ is the vocabulary of the grammar, made of the union of terminal (Σ) and nonterminal symbols (N). In case a grammar does not fit in such characteristics the algorithm stops (such grammars will be hereafter referred as NLALR grammars). If continuation is performed, Charles states that there might be a chance that termination will never be reached.

5. SAIDE’s Internal Algorithms

This section presents the core algorithms we propose to support the methodology discussed in Section 3. For those already discussed in [Passos et al. 2007], no presentation is made. The algorithms are organized in three parts: conflict removal, conflict listing and table compression.
5.1. Conflict Removal

Charles calculates the lookaheads necessary to extend a given token by simulating the steps of the LR(0) automaton. The steps used in such computation detects NLALR grammars. When such detection occurs, execution is immediately stopped. At this point, there might exist some non-solved conflicts that will not be reported to the user, resulting in an incorrect number of reported conflicts. This gives a wrong impression of the total amount of conflicts and disables SAIDE’s capacity to determine the order in which conflicts should be removed. In this manner, SAIDE’s methodology as originally proposed becomes inapplicable.

SAIDE overcomes this by establishing the whole set of non-solved conflicts even in the presence of NLALR grammars. It uses six algorithms, that are the result of modifications over the ones originally proposed by Charles.

The whole process of solving conflicts starts with SWEEP, shown in Figure 3. The SWEEP algorithm must be called for each state \( p \) that contains a conflict. This procedure inspects the number of entries in the pairs \( (p, a) \) of the action matrix, where \( a \) is a terminal symbol. If \( (p, a) \) has cardinality greater than two, it determines all possible stacks (sources), given the initial stack \( [p] \), that result in reading \( a \). If \( (p, a) \) contains a shift action, then \( [p] \) is a valid stack. Reduce actions must also be considered. If a reduction by \( A \to \omega \) belongs to \( (p, a) \), then the start stack is one made of the predecessor state of \( p \) under \( \omega \). Function PRED returns such predecessor state. In both cases, the stacks are stored in the sources dictionary as part of pairs \( (stk, w) \), where \( w \) corresponds to the string of tokens that would have been read by the LR(0) automaton having the stack \( stk \). Calling FOLLOW-SOURCES\(_A\) completes the initial set of pairs \( (stk, w) \). After the definition of all sources, RESOLVE-CONFLICTS is executed in an attempt to remove the conflicts in \( (p, a) \).

The procedure FOLLOW-SOURCES\(_A\), shown in Figure 4, is a façade procedure to FOLLOW-SOURCES\(_B\). Five arguments are received: the set of pairs \( (stack, w) \) – stored in sources, the current stack, the current symbol \( X \in V \) denoting the transition that must be performed from the top state of the current stack, \( a \) marking the terminal symbol that has to be eventually found as a transition symbol, and \( w \) as the current processed string. Before FOLLOW-SOURCES\(_B\) starts simulating the LR(0) steps in order to find all stacks that will lead to the reading of \( a \), FOLLOW-SOURCES\(_A\) puts the current stack in a tree format. The idea of using a tree structure comes from [Kristensen and Madsen 1981] and is used as a way to prevent infinite loop. Each node in the tree stores, besides its list of children, a pair \( (state, z) \) as its value. Given a node \( n \), the string \( z \) stands as the string processed by the LR(0) automaton given the states from the root of the tree to the node \( n \). In the built tree, each node has a unique numeric identifier. Instantiating nodes using NODE controls such uniqueness. The value and the identifier of a node can be retrieved at any time by calling VALUE and ID, respectively.

FOLLOW-SOURCES\(_B\), presented in Figure 5, aims to find from a given initial stack, encoded as a path in the tree of states, the set of stacks that will lead to the reading of \( a \). Eight arguments must be received: the set of sources, as in FOLLOW-SOURCES\(_A\), the current transition to be performed as part of the LR(0) simulation – encoded as a triple

---

3The functions FST and SND will be used to retrieve the first and the second component of a pair.

4The sources dictionary is indexed by parsing actions.
The terminal \( a \), the current processed string \( w \), the root of the tree, the current node tree and the sets \( \text{visited} \) and \( \text{roots} \). When \( a \) appears as a transition symbol, the procedure stores in \( \text{sources} \) the pair \((\text{stack},wa)\), where \( \text{stack} \) is given by the states in the values of the nodes in the path in the tree from \( \text{root} \) to \( \text{node} \). The first part of loop control in \( \text{FOLLOW-SOURCES}_B \) is located in lines 2–8. At that point, each time the procedure is activated, \( \text{FOLLOW-SOURCES}_B \) checks if the current stack and transition were visited in a past time. Instead of storing the whole sequence of states on the stack, it is enough to know the current node’s identifier, for it also defines a unique stack. For optimization purposes, the pairs \((\text{id},\text{transition})\) are stored in two distinct sets: \( \text{roots} \) and \( \text{visited} \). The \( \text{roots} \) set is only used for stacks whose size is one. The size of the current stack is obtained by calling \( \text{HEIGHT} \), which returns the height from the root node to the current node. Note that if the size is one, storing the node’s identifier is not necessary, since the root state corresponds to the source state in the \( \text{transition} \) triple. For stacks greater than one in size, \( \text{visited} \) is used. Lines 10–24 inspect all edges leaving \( q \), where \( q \) is the destination state of the \( \text{transition} \) parameter. Two possibilities arise: a transition over a nullable symbol or a terminal symbol. From the first we must continue the simulation by pushing \( q \) onto the stack. To do this, we add a new child node to the current node’s children list. The created node stores a pair \((q,w)\) as its value. Next, \( \text{FOLLOW-SOURCES}_B \) is recursively called. The second situation occurs when there is a transition symbol from \( q \) that matches \( a \). In this case, the current stack is retrieved by getting the inverse order of the states from the current node to the root of the tree, and the pair \((\text{stack},wa)\) is added to \( \text{sources} \). Again, loop control is performed in the 10th line. To aid this control, we use the \( \text{GET-FROM} \) function. Given a node \( n_x \) and a value \( v \), \( \text{GET-FROM} \) returns a non-null reference to a node \( n_y \) in the path from \( n_x \) to the root of the tree whose value coincide with \( v \). If so, pushing \( q \) onto the stack results in a cycle: \([(p_1p_2...p_nq),w]^*\) \(\rightarrow\) \([(p_1p_2...p_nq_1q_2...q_nq),w]\). The lines 26–37 are responsible for making reductions. Reductions while simulating LR(0) may lead to underflow situations, i.e., the act of popping more states than currently available on the stack. If \( |\gamma_1\gamma_2| \) states should be popped, but only \( |\gamma_2| \) are available on the stack, being \( \gamma_1\gamma_2 \) the right hand side of a production, the predecessor state of the \( \gamma_1 \) is retrieved and put as the top element of an unitary stack, used as a parameter to a recursive call.

The procedure \( \text{RESOLVE-CONFLICTS} \), shown in Figure 6, checks if a conflict is removed. If not, it extends the DFA in another level of lookaheads, respected \( k_{\text{max}} \). Four arguments are mandatory: a state \( q \) containing conflicts under \( t \), also a received parameter, the \( \text{sources} \) dictionary (as in \( \text{Sweep} \) and \( \text{FOLLOW-SOURCES}_A \)) and \( n \), the number of lookaheads used so far. Its execution starts checking if \( n \) is greater than \( k_{\text{max}} \) or \( t \) is the EOF marker. In either case, the extension of lookaheads should not go further and the procedure returns. Otherwise, a new line \( p \) is allocated in the action table, after its last line, and each entry in \( p \) is given the empty set. Later, the conflict entry \((q,t)\) points to \( p \) by a lookahead action – \( Lp \). Next, for each action indexing \( \text{sources} \), each source \((\text{stack},w)\) is inspected. Each token that can follow \( t \), given \( \text{stack} \), is calculated by calling \( \text{NEXT-LOOKAHEADS}_A \). For each returned token \( a \), the appropriate actions are put into \((p,a)\) in the action table. After determining the values in \( p \)'s entries, for the entries whose cardinality is greater than one, the conflict removal process continues by calling \( \text{FOLLOW-SOURCES}_A \). This is necessary, since \( \text{NEXT-LOOKAHEADS}_A \) returns the tokens that can extend \( t \), but not the context in which they were obtained (source stacks).
Analogous to FOLLOW-SOURCES\(_A\), NEXT-LOOKAHEADS\(_A\), shown in Figure 7, is a façade procedure to NEXT-LOOKAHEADS\(_B\). It structures the received stack in a tree format.

Having such tree, the procedure NEXT-LOOKAHEADS\(_B\), presented in Figure 8, fast calculates the tokens that can be found given a stack and a transition. To achieve this, it uses two external functions defined in [DeRemer and Pennello 1982]: READ\(_1\) and FOLLOW\(_1\). From a state \(p\) and a symbol \(X\), READ\(_1\) returns the tokens that can be read from \(\text{GOTO}_0(p, X)\) either directly or under nullable transitions; FOLLOW\(_1\) returns the tokens either in READ\(_1\)(\(p, X\)) or in FOLLOW\(_1\)(\(p_0, C\)), as long as \(C \rightarrow \alpha\bullet X\beta \in p, \beta \Rightarrow \lambda\) and \(p_0 \in \text{PRED}(p, \alpha)\). Given a stack and a transition, NEXT-LOOKAHEADS\(_B\) grabs all tokens returned by READ\(_1\), i.e., the tokens that can be read given the current context. Reductions are treated as in FOLLOW-SOURCES\(_B\), except that in cases of underflow, the simulation does not go further. Instead, the algorithm retrieves the desired tokens by calling FOLLOW\(_1\). To reduce under non-underflow cases, the procedure pops states by calling UP; given a node \(n\) and a value \(k\), UP returns the \(k\)-th ancestor of \(n\). By using READ\(_1\) and FOLLOW\(_1\), which can be precomputed, NEXT-LOOKAHEADS\(_B\) does not have to search for source stacks when looking for lookaheads. The procedure’s loop control is achieved just like FOLLOW-SOURCES\(_B\), i.e., by storing all visited transitions.

The presented algorithms do not have the limitation of stopping when dealing with NLALR grammars; termination is guaranteed to be reached under any circumstances.

```
SWEEP(p)
  1 for a ∈ Σ
  2 do if |Action[p, a]| > 0 ∧ a ≠ $
  3 then sources ← Ø
  4 for act ∈ Action[p, a]
  5 do if act is a shift action
  6 then sources[act] ← {(p, a)}
  7 else ASSERT(act = reduce by rule A → α)
  8 for p₀ ∈ PRED(p, α)
  9 do FOLLOW-SOURCES\(_A\)(sources[act], [p₀], A, a, λ)
 10 RESOLVE-CONFLICTS(p, a, sources, 2)
```

Figure 3. Main procedure to automatically remove conflicts.

5.2. Conflict Listing

When using LALR\((k_v)\) action tables, the parser generator must not miscalculate the number of remaining conflicts. To illustrate this, consider the following table when \(k_{\text{max}} = 1\):

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>S11, R8</td>
<td>R3, R8</td>
<td>R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using \(k_{\text{max}} = 2\), the table is given by:

5Originally, READ\(_1\) and FOLLOW\(_1\) were defined over nonterminal transitions - in NEXT-LOOKAHEADS\(_B\), READ\(_1\), as defined by Charles, was generalized to terminal and nonterminal transitions.
FOLLOW-SOURCES$_A(sources, stack, X, a, w)$

1. ASSERT(stack = [p$_1$...p$_n$])
2. root ← node ← NODE((p$_1$, λ))
3. for 2 ≤ i ≤ n
4. do node$_2$ ← NODE((p$_i$, λ))
5. ADD-CHILD(node, node$_2$)
6. node ← node$_2$
7. SND(VALUE(node)) ← w
8. FOLLOW-SOURCES$_B(sources, (p$_n$, X, GOTO$_0$(p$_n$, X)), a, w, root, node, ∅, ∅)

Figure 4. Façade procedure FOLLOW-SOURCES$_A$.

|   | a | b | c | d | e  | f | $|$ |
|---|---|---|---|---|----|---|----|
| 8 |   |   |   |   | L13| L14|R3  |
|...|   |   |   |   |    |    |    |
|13 |   |   |   |   | S11|R8  |    |
|14 |   |   |   | R3| R8 | R3 | R8 |

Simply examining the number of conflictuous entries in the latter table allows identifying four conflicts, instead of the original two. When using $k_{\text{max}}$ ≥ 2, SAIDE performs a depth first search from the lookahead action in an entry $(p, a)$, and retrieves the actions in the entries that still contain conflicts. The obtained set of actions $\text{acts}$ is a subset of the original set of conflicts. When performing the search, SAIDE also keeps track of all traversed edges of the DFA, and thus obtain the strings of length up to $k_{\text{max}}$ for which the conflict is not removed. The number of conflicts for $(p, a)$ is given by:

$$\left(\left|\text{shift}\right| \times \left|\text{reds}\right|\right) + \left(\lambda x. \text{if } x \geq 2 \text{ then } 1 \text{ else } 0\right)\left|\text{reds}\right|$$

where

$$\begin{align*}
\text{shift} &= \{s \mid s \in \text{acts} \land s \text{ is a shift action}\} \\
\text{reds} &= \{r \mid r \in \text{acts} \land r \text{ is a reduce action}\}
\end{align*}$$

For the conflicts that could not be automatically removed, SAIDE lists them in a heuristic manner. The heuristic here discussed builds a conflict graph, whose vertexes represent LALR states with at least one non-solved conflict. A directed edge connects $p$ to $q$ if there is a path from $p$ to $q$ in the LALR automaton, i.e., $p$ propagates lookaheads to $q$. From this graph, a second one is built, formed by the SCCs of the first. In this graph, a directed edge between two vertexes exists if at least one vertex in the first SCC connects to another vertex in the second SCC. The SCCs graph is then topologically sorted. From the obtained graph, the conflicts are listed according to the order of the SCCs, from left to right. Given an SCC $c$, all conflicts in state $p \in c$ are put on the listing.

5.3. Table Compression

An important problem using LALR($k$) parsers is due to space requirements, for the size of the action table substantially grows when the value of $k$ increases.
FOLLOW-SOURCES\(_B\)(sources, transition, a, w, root, node, visited, roots)

1. stackSize ← HEIGHT(node, root)
2. if stackSize = 1
   3. then if transition ∈ roots
       4. then return
       5. else roots ← roots ∪ \{transition\}
6. else if (ID(node), transition) ∈ visited
   7. then return
8. else visited ← visited ∪ \{(ID(node), transition)\}

9. ASSERT(transition = (ts, X, q))

10. for Y ∈ V | GOTO\(_0\)(q, Y) is defined
11. do if Y ⇒ λ ∧ GET-FROM(node, (q, w)) = nil
12. then node\(_2\) ← NODE((q, w))
13. ADD-CHILD(node, node\(_2\))
14. FOLLOW-SOURCES\(_B\)
15. (sources, (q, Y, GOTO\(_0\)(q, Y)), a, w, root, node\(_2\), visited, roots)
16. else if Y = a
   17. then node\(_2\) ← node
18. list ← [FST(VALUE(node\(_2\)))]
19. while \(\text{node}_2 \leftarrow \text{PARENT}(\text{node}_2)\) \(\neq \text{nil}\)
20. do list ← list + [FST(VALUE(node\(_2\)))]
21.
22. ASSERT(list = \([p_n...p_1]\))
23. stack ← \([p_1...p_0q]\)
24. sources ← sources ∪ \{(stack, wa)\}

25. bottom ← FST(VALUE(root))
26. for C → γ • ∈ ts | C \(\neq S\)
27. do if |γ| + 1 < stackSize
28. then node\(_2\) ← UP(node, |γ|)
29. SND(VALUE(node\(_2\))) ← w
30. ts\(_2\) ← FST(VALUE(node\(_2\)))
31. FOLLOW-SOURCES\(_B\)
32. (sources, (ts\(_2\), C, GOTO\(_0\)(ts\(_2\), C)), a, w, root, node\(_2\), visited, roots)
33. else ASSERT(γ = GOTO\(_0\)(ts\(_2\), C)), a, w, root, node\(_2\), visited, roots)
34. for p\(_0\) ∈ PRED(bottom, γ\(_1\))
35. do root\(_2\) ← NODE((p\(_0\), w))
36. FOLLOW-SOURCES\(_B\)
37. (sources, (p\(_0\), C, GOTO\(_0\)(p\(_0\), C)), a, w, root\(_2\), root\(_2\), visited, roots)

Figure 5. Procedure FOLLOW-SOURCES\(_B\).

In LALR(\(k_v\)) parsers, the number of inspected lookaheads is minimized, but the size of action parsing tables are still considerable. We performed tests with nine LALR(1) compression techniques (ACS [Aho et al. 1986], BCS [Bigonha and Bigonha 1983], RDS [Dencker et al. 1984], SDS [Beach 1974], RCS [Tewarson 1968], SZS [Dencker et al. 1984], GCS [Schmitt 1979], LES [Bell 1974], ...
RESOLVE-CONFLICTS(q, t, sources, n)
1 if t = $ \lor n > k_{\text{max}}$
2 then return
3 allocate a new line p in the action table
4 for a ∈ Σ
5 do Action[p, a] ← ∅
6 Action[q, t] ← {Lp}
7 for act indexing sources
8 do for src ∈ sources[act]
9 do ASSERT(src = (stack, w))
10 la ← NEXT-LOOKAHEADS_A(stack, t)
11 for a ∈ la
12 do Action[p, a] ← Action[p, a] ∪ {act}
13 for a ∈ (Σ − {$\}$) \ Action[p, a] > 1
14 do for act ∈ Action[p, a]
15 do nSources ← ∅
16 for src ∈ sources[act]
17 do ASSERT(src = (stack, w))
18 FOLLOW-SOURCES_A(nSources[act], stack, t, a, w)
19
20 RESOLVE-CONFLICTS(p, a, nSources, n + 1)

Figure 6. Procedure RESOLVE-CONFLICTS.

RMS [Passos 2007]) using the grammars of C, C#, HTML, Java and VB programming languages. The result of this study is directly generalized to LALR(k) parsers, since they have the same layout as LALR(1) action parsing tables.

From the experiment, BCS and the combination of GCS, LES and RMS presented the highest compression rates, respectively 95% and 87%, in average. For BCS, there is the price of the overhead caused by the substitution of a direct access by a linear access to an interval of an array containing syntactic actions. In this scheme, the generated parser might execute unnecessary reductions, although correctness is preserved. The combination of GCS, LES and RMS preserves O(1) access time and guarantees execution of the same number of actions as the non-compressed parser.

6. Experimental results

We performed tests in order to evaluate the automatic conflict mechanism and how the proposed loop control impacts in execution time. Table 1 shows the results obtained for the Algol-60, Scheme, Oberon-2 and Notus original grammars.

From the experiment, the number of conflicts was reduced in 51% and 97% in Scheme and Oberon-2 when using $k_{\text{max}} = 2$ in comparison with one token ahead. In Algol-60 the number of conflicts is not affected at any time; in Notus, there is a reduction in 6% when using at most two lookaheads. From $k_{\text{max}} \geq 3$, there’s practically no overall change in the number of conflicts. The discrepancy between Scheme and Oberon-2 when compared to Notus and Algol-60 is that these are more human readable than closer to LALR
NEXT-LOOKAHEADS\(_A\)(stack, \(t\))
1 \(la \leftarrow \emptyset\)
2 ASSERT(stack = \([p_1 ... p_n]\))
3 root \(\leftarrow node \leftarrow\) NODE(\(p_1\))
4 for \(2 \leq i \leq n\) do
5 \(node_2 \leftarrow\) NODE(\(p_i\))
6 ADD-CHILD(node, node\(_2\))
7 node \(\leftarrow node_2\)
8 NEXT-LOOKAHEADS\(_B\)(la, \((p_n, t, \text{GOTO}_0(p_n, t))\), root, node, \(\emptyset\))
9 return \(la\)

Figure 7. Façade procedure NEXT-LOOKAHEADS\(_A\).

<table>
<thead>
<tr>
<th>Grammar</th>
<th>(k_{\text{max}} = 1)</th>
<th>(k_{\text{max}} = 2)</th>
<th>(k_{\text{max}} = 3)</th>
<th>(k_{\text{max}} = 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol-60</td>
<td>61</td>
<td>670</td>
<td>61</td>
<td>2,480</td>
</tr>
<tr>
<td>Scheme</td>
<td>78</td>
<td>839</td>
<td>38</td>
<td>1,320</td>
</tr>
<tr>
<td>Oberon-2</td>
<td>32</td>
<td>898</td>
<td>1</td>
<td>1,464</td>
</tr>
<tr>
<td>Notus</td>
<td>575</td>
<td>1,068</td>
<td>541</td>
<td>34,828</td>
</tr>
</tbody>
</table>

Table 1. Evaluation of increasing \(k\) in automatic conflict removal.

conformance. The opposite situation occurs with Scheme and Oberon-2.

As a case study, the tool was used in building a parser for the Machña programming language [Bigonha et al. 2007]. The writing of its syntax specification was incrementally performed. An increment is the result of adding new rules to the previous increment when the set of conflicts of the latter becomes empty. In the experiment, \(k_{\text{max}} = 2\) was used. At each reported conflict, the four phases for manual removal were applied. The application of the four phases defines a step. Figure 9 shows the dot graph for the number of conflicts obtained in each step. For reading purposes, the dots were connected to better identify the increasing and decreasing of conflicts. In the presented graph, the frontier between increments is marked by a dotted line. With exception to increment four, all steps inside other increments showed a decrease in the number of conflicts. The increase of conflicts occurs in the border of two increments, which is expected due to the adding of new rules. The produced parsing table was compressed with BCS and the combination of GCS, LES and RMS, resulting in 98% and 94% of compression rate.

7. Conclusion

This article presented an LALR parser generator supporting conflict resolution. Among the contributions of our work, we highlight the following:

- the process of conflict removal is eased by automatic conflict removal. In particular, the present algorithms remove some conflicts caused by lack of right context;
- for the cases in which manual removal is required, the tool assists users through a well defined methodology;
NEXT-LOOKAHEADS_B(la, transition, root, node, visited)
1  ASSERT(transition = (ts, X, q))
2  bottom ← VALUE(root)
3  if (ID(node), transition) ∈ visited then return
4    la ← la ∪ READ_1(ts, X)
5  stackSize ← HEIGHT(node, root)
6  nStacks ← ∅
7  for C → γ • Xδ | δ* ⇒ λ ∧ C ≠ S
8    do if |γ| + 1 < stackSize
9      then node_2 ← UP(node, |γ|)
10     nStacks ← nStacks ∪ {(node_2, C)}
11    else ASSERT(γ = γ_1γ_2), where |γ_2| = stackSize − 1
12       for p_0 ∈ PRED(bottom, γ_1)
13         do la ← la ∪ FOLLOW_1(p_0, C)
14  for (n, C) ∈ nStacks
15    do ts ← VALUE(n)
16  NEXT-LOOKAHEADS_B(la, (ts, C, GOTO_0(ts, C)), root, n, visited)

Figure 8. Procedure NEXT-LOOKAHEADS_B.

• modifications in Charles’ proposal in order to accept any context free grammar.
  This makes the methodology independent of grammar characteristics;
• generalization of LALR(1) compression schemes to LALR(k_v) approach, which
  turns the variable k approach viable in the sense that memory requirements are
  minimized.

All obtained results are based on empirical data, determined by using established and also
new programming languages, such as Notus and Machina. The presented results indicate
that the application of the methodology contributes to the constant decrease on the number
of conflicts in a grammar and that, in general, the time spent in calculating lookaheads in
any context free grammar does not incur in a major overhead given today CPUs clocks.

References

Addison-Wesley.


parsing tables. Technical report, Departament of Computer Science, Federal University
of Minas Gerais.
Figure 9. Conflict removal graph for the Machina programming language.


