Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3447-3464
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © J.UCS

# An LALR Parser Generator Supporting Conflict Resolution

**Leonardo Teixeira Passos**
(Federal University of Minas Gerais, Belo Horizonte, Brazil
leonardo@dcc.ufmg.br)

**Mariza A. S. Bigonha**
(Federal University of Minas Gerais, Belo Horizonte, Brazil
mariza@dcc.ufmg.br)

**Roberto S. Bigonha**
(Federal University of Minas Gerais, Belo Horizonte, Brazil
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.

**Key Words:** lalr parsing, automatic conflict removal, table compression, methodology
**Category:** D.3.4, F.4.2

## 1 Introduction

The usual way to build a bottom-up syntax analyzer 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 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 $k = max(\{k'$, *such that* $k'$ *is the number of lookaheads needed to solve a given conflict among all conflicts reported by the parser generator and* $k$ *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 discussion of all types of conflicts can be found in [Passos et al. 2007].

[1] This test was performed using the grammar provided in `http://www.devincook.com/goldparser/grammars/index.html`.

## 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 intervention. 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 removed. A conflict must be listed before those that appear as a consequence of the existence 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.

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.

---

[2] SAIDE /saɪd/: Syntax Analyzer Integrated Development Environment.

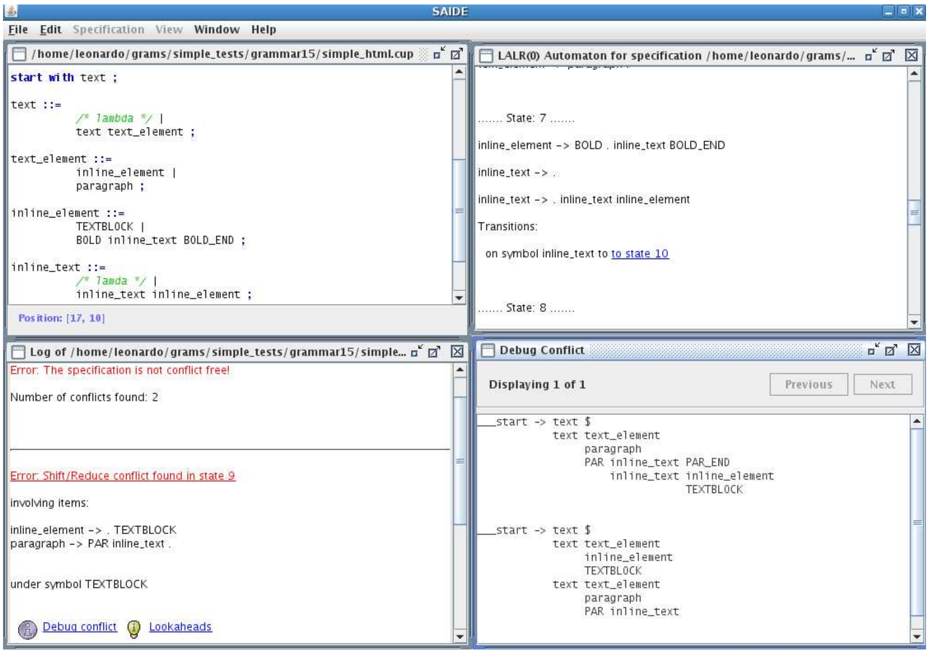


**Figure 1:** SAIDE's main window.

## 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 $k_{max}$. 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 lookaheads consist of the tokens that may follow $t_i$, given a conflictual entry $(q, t_i)$. This process continues until the conflicts are removed or the depth of the DFA reaches $k_{max}$. 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 $k_{max}$ 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 $k_{max}$ was three, in $q_1$ only two lookaheads were used.

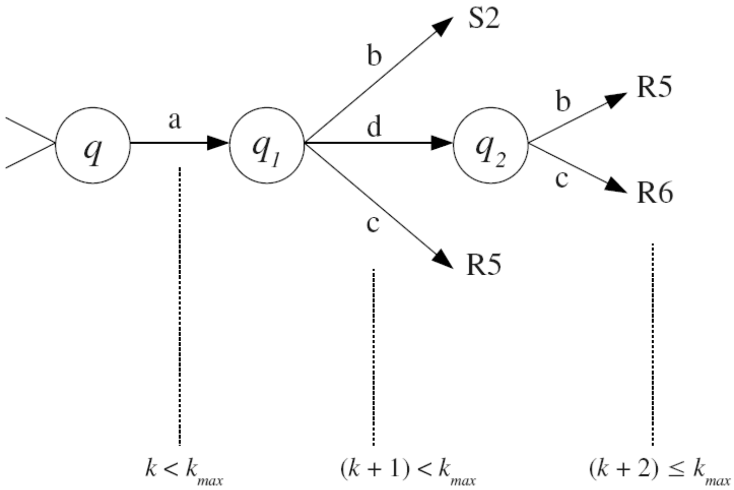


Figure 2: DFA to solve a given conflict by extending the number of lookaheads respected the limit $k_{max} = 3$.

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* – $Ln$, 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:

$$\begin{aligned} bnf &\to rlist \\ rlist &\to rlist\ rule \\ &\mid\ \lambda \\ rule &\to \mathbf{s}\ \text{“}\!\to\!\text{”}\ slist \\ slist &\to slist\ \mathbf{s} \\ &\mid\ \lambda \end{aligned}$$

This grammar is not LALR(1); it is LALR(2). The following action parsing table results from the LALR($k_v$) approach:

| *Action* | | **s** | “→” | “\$” |
|---|---|---|---|---|
| | 0 | R3 | | R3 |
| | 1 | S3 | | R1 |
| | 2 | | | ACC |
| | 3 | | S5 | |
| | 4 | R2 | | R2 |
| | 5 | R6 | | R6 |
| | 6 | L8 | | R4 |
| | 7 | R5 | | R5 |
| *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 a relation between transitions [3] [DeRemer and Pennello 1982]:

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

[3] $(p, A)$ denotes a transition under $A$ in state $p$ in the LR(0) automaton.

where $\text{GOTO}_0$ is the transition function of the LR(0) 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 ($\Sigma$) 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 of entering infinite loop.

## 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. His algorithms stop as soon as the grammar is found to be NLALR, ignoring the possibility of having non-inspected automatic solvable conflict entries in the action table. If such entries exist, given the LALR($k_v$) approach, users face a number of conflicts that is greater than the real quantity. In addition, they may spend time manually removing conflicts that were supposed to disappear automatically. In this scenario, SAIDE's methodology as originally proposed becomes inapplicable.

SAIDE overcomes this by establishing the genuine set of conflicts that cannot be solved. It uses six algorithms that are the result of modifications over the ones originally proposed by Charles. Due to a proper infinite loop control, all proposed algorithms are guaranteed to terminate, even in the presence of NLALR grammars.

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 $(p, a)$[4] 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 $A \rightarrow \omega$ belongs to $(p, a)$, then the start stack is one made of a predecessor state of $p$ under $\omega$. Function PRED returns such predecessor states. In both cases, the stacks are stored in the *sources* dictionary[5] as part of pairs $(stk, w)$, where $w$ corresponds to the string of

[4] The functions FST and SND will be used to retrieve the first and the second component of a pair.
[5] The *sources* dictionary is indexed by parsing actions.

```
SWEEP(p)
for a ∈ Σ
do if |Action[p, a]| > 0 ∧ a ≠ $
      then sources ← ∅
           for act ∈ Action[p, a]
           do if act is a shift action
                 then sources[act] ← {([p], a)}
                 else ASSERT(act = reduce by rule A → α)
                      for p_0 ∈ PRED(p, α)
                      do FOLLOW-SOURCES_A(sources[act],
                            [p_0], A, a, λ)
           RESOLVE-CONFLICTS(p, a, sources, 2)
```

**Figure 3:** Main procedure to automatically remove conflicts.

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 – structured as a triple (*source-state*, *transition-symbol*, *desti-*

*nation-state*), the terminal $a$, the current processed string $w$, the root of the tree, the current node tree and the sets *visited* and *roots*. When $a$ appears as a transition symbol, the procedure stores in *sources* the pair $(stack, wa)$, where *stack* is given by the states in the values of the nodes in the path in the tree from *root* to *node*. The first part of infinite loop control in FOLLOW-SOURCES$_B$ is located in lines $2-8$. At that point, each time the procedure is activated, 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 $(id, transition)$ are stored in two distinct sets: *roots* and *visited*. The *roots* set is only used for stacks whose size is one. The size of the current stack is obtained by calling 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 *transition* triple. For stacks greater than one in size, *visited* is used. Lines $10-25$ inspect all edges leaving $q$, where $q$ is the destination state of the *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, 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 $(stack, wa)$ is added to *sources*. Again, infinite loop control is performed in the $11^{\text{th}}$ line. To aid this control, we use the GET-FROM function. Given a node $n_x$ and a value $v$, GET-FROM returns a non-nullable reference to a node $n_y$ in the path from $n_x$ (inclusively) to the root of the tree whose value coincide with $v$. If so, pushing $q$ onto the stack results in a cycle: $([p_1p_2...p_nq], w) \overset{*}{\vdash} ([p_1p_2...p_nqq_1q_2...q_nq], w)$. The lines $27-40$ 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 RESOLVE-CONFLICTS, shown in Figure 6, checks if a conflict is removed. If not, it extends the DFA in another level of lookaheads, respected $k_{max}$. Four arguments are mandatory: a state $q$ containing conflicts under $t$, also a received parameter, the *sources* dictionary (as in SWEEP and FOLLOW-SOURCES$_A$) and $n$, the number of lookaheads used so far. Its execution starts checking if $n$ is greater than $k_{max}$ or $t$ is the EOF marker. In

FOLLOW-SOURCES$_A$(*sources*, *stack*, $X$, $a$, $w$)
  ASSERT(*stack* $= [p_1 ... p_n]$)
  *root* $\leftarrow$ *node* $\leftarrow$ NODE($(p_1, \lambda)$)
  **for** $2 \le i \le n$
  **do** $node_2 \leftarrow$ NODE($(p_i, \lambda)$)
    ADD-CHILD(*node*, $node_2$)
    *node* $\leftarrow node_2$
  SND(VALUE(*node*)) $\leftarrow w$
  FOLLOW-SOURCES$_B$(*sources*, $(p_n, X, \mathrm{GOTO}_0(p_n, X)), a, w,$
  *root*, *node*, $\emptyset, \emptyset$)

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

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 *sources*, each source (*stack*, $w$) is inspected. Each token than can follow $t$, given *stack*, is calculated by calling 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 FOLLOW-SOURCES$_A$. This is necessary, since 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 the external functions READ$_1$ and FOLLOW$_1$, both defined in [DeRemer and Pennello 1982]. From a state $p$ and a symbol $X$, READ$_1$ returns the tokens that can be read from 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 $C \rightarrow \alpha \bullet X\beta \in p$, $\beta \overset{*}{\Rightarrow} \lambda$ and $p_0 \in \mathrm{PRED}(p, \alpha)$[6]. 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 un-

[6] Originally, READ$_1$ and FOLLOW$_1$ were defined over nonterminal transitions - in NEXT-LOOKAHEADS$_B$, READ$_1$, as defined by Charles, was generalized to terminal an nonterminal transitions.

derflow, the simulation does not go further. Instead, the algorithm retrieves the desired tokens by calling $\text{FOLLOW}_1$. 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 $\text{READ}_1$ and $\text{FOLLOW}_1$, which can be precomputed, NEXT-LOOKAHEADS$_B$ does not have to search for source stacks when looking for lookaheads. The procedure's infinite 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.

### 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 $k_{max} = 1$:

| | **a** | **b** | **c** | **d** | **e** | **f** | **$** |
|---|---|---|---|---|---|---|---|
| 8 | | | | | S11, R8 | R3, R8 | R3 |
| ... | | | | | | | |

Using $k_{max} = 2$, the table is given by:

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

Simply examining the number of conflictual entries in the latter table allows identifying four conflicts, instead of the original two. When using $k_{max} \geq 2$, 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 *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 $k_{max}$ for which the conflict is not removed. The number of conflicts for $(p, a)$ is given by:

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

where

$$\begin{aligned} \mathit{shift} &= \{s \mid s \in \mathit{acts} \wedge\ s\ \mathit{is\ a\ shift\ action}\} \\ \mathit{reds} &= \{r \mid r \in \mathit{acts} \wedge\ r\ \mathit{is\ a\ reduce\ action}\} \end{aligned}$$

```
FOLLOW-SOURCES_B(sources, transition, a, w, root, node, visited, roots)
stackSize ← HEIGHT(node, root)
if stackSize = 1
   then if transition ∈ roots
           then return
           else roots ← roots ∪ {transition}
   else if (ID(node), transition) ∈ visited
           then return
           else visited ← visited ∪ {(ID(node), transition)}
ASSERT(transition = (ts, X, q))
for Y ∈ V | GOTO_0(q, Y) is defined
do if Y ⇒* λ ∧ GET-FROM(node, (q, w)) = nil
      then node_2 ← NODE((q, w))
           ADD-CHILD(node, node_2)
           FOLLOW-SOURCES_B
           (sources, (q, Y, GOTO_0(q, Y)), a, w,
           root, node_2, visited, roots)
      else if Y = a
              then node_2 ← node
                   list ← [FST(VALUE(node_2))]
                   while (node_2 ← PARENT(node_2)) ≠ nil
                   do list ← list + [FST(VALUE(node_2))]

                   ASSERT(list = [p_n...p_1])
                   stack ← [p_1...p_n q]
                   sources ← sources ∪ {(stack, wa)}
bottom ← FST(VALUE(root))
for C → γ• ∈ ts | C ≠ S
do if |γ| + 1 < stackSize
      then node_2 ← UP(node, |γ|)
           SND(VALUE(node_2)) ← w
           ts_2 ← FST(VALUE(node_2))
           FOLLOW-SOURCES_B
           (sources, (ts_2, C, GOTO_0(ts_2, C)),
           a, w, root, node_2, visited, roots)
      else ASSERT(γ = γ_1 γ_2), where |γ_2| = stackSize − 1
           for p_0 ∈ PRED(bottom, γ_1)
           do root_2 ← NODE((p_0, w))
              FOLLOW-SOURCES_B
              (sources, (p_0, C, GOTO_0(p_0, C)), a, w, root_2,
              root_2, visited, roots)
```

**Figure 5:** Procedure FOLLOW-SOURCES$_B$.

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

```
RESOLVE-CONFLICTS(q, t, sources, n)
if  t = $ ∨ n > k_max
    then  return
allocate a new line p in the action table
for  a ∈ Σ
do  Action[p, a] ← ∅
Action[q, t] ← {Lp}
for  act indexing sources
do  for  src ∈ sources[act]
        do  ASSERT(src  = (stack, w))
            la ← NEXT-LOOKAHEADS_A(stack, t)
            for  a ∈ la
            do  Action[p, a] ← Action[p, a] ∪ {act}
for  a ∈ (Σ − {$}) | |Action[p, a]| > 1
do  for  act ∈ Action[p, a]
        do  nSources ← ∅
            for  src ∈ sources[act]
            do  ASSERT(src = (stack, w))
                FOLLOW-SOURCES_A(nSources[act], stack, t, a, w)

        RESOLVE-CONFLICTS(p, a, nSources, n + 1)
```

**Figure 6:** Procedure RESOLVE-CONFLICTS.

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.

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 schemes [7]:

- the compression scheme proposed in [Aho et al. 1986] (ACS);

[7] The authors of ACS and BCS didn't name their methods. For reference purposes, the adopted names reflect the corresponding bibliography entries in this work.

$\text{NEXT-LOOKAHEADS}_A(\mathit{stack}, t)$
$\quad \mathit{la} \leftarrow \emptyset$
$\quad \text{ASSERT}(\mathit{stack} = [p_1 ... p_n])$
$\quad \mathit{root} \leftarrow \mathit{node} \leftarrow \text{NODE}(p_1)$
$\quad$ **for** $2 \leq i \leq n$
$\quad$ **do** $\mathit{node}_2 \leftarrow \text{NODE}(p_i)$
$\qquad \text{ADD-CHILD}(\mathit{node}, \mathit{node}_2)$
$\qquad \mathit{node} \leftarrow \mathit{node}_2$
$\quad \text{NEXT-LOOKAHEADS}_B(\mathit{la}, (p_n, t, \text{GOTO}_0(p_n, t)), \mathit{root}, \mathit{node}, \emptyset)$
$\quad$ **return** $\mathit{la}$

**Figure 7:** Façade procedure $\text{NEXT-LOOKAHEADS}_A$.

$\text{NEXT-LOOKAHEADS}_B(\mathit{la}, \mathit{transition}, \mathit{root}, \mathit{node}, \mathit{visited})$
$\quad \text{ASSERT}(\mathit{transition} = (ts, X, q))$
$\quad \mathit{bottom} \leftarrow \text{VALUE}(\mathit{root})$
$\quad$ **if** $(\text{ID}(\mathit{node}), \mathit{transition}) \in \mathit{visited}$
$\qquad$ **then return**
$\quad \mathit{la} \leftarrow \mathit{la} \cup \text{READ}_1(ts, X)$
$\quad \mathit{stackSize} \leftarrow \text{HEIGHT}(\mathit{node}, \mathit{root})$
$\quad \mathit{nStacks} \leftarrow \emptyset$
$\quad$ **for** $C \rightarrow \gamma \bullet X\delta \in ts \mid \delta \overset{*}{\Rightarrow} \lambda \wedge C \neq S$
$\quad$ **do if** $|\gamma| + 1 < \mathit{stackSize}$
$\qquad\quad$ **then** $\mathit{node}_2 \leftarrow \text{UP}(\mathit{node}, |\gamma|)$
$\qquad\qquad\quad \mathit{nStacks} \leftarrow \mathit{nStacks} \cup \{(\mathit{node}_2, C)\}$

$\qquad\quad$ **else** $\text{ASSERT}(\gamma = \gamma_1\gamma_2),\ \mathit{where}\ |\gamma_2| = \mathit{stackSize} - 1$
$\qquad\qquad\quad$ **for** $p_0 \in \text{PRED}(\mathit{bottom}, \gamma_1)$
$\qquad\qquad\quad$ **do** $\mathit{la} \leftarrow \mathit{la} \cup \text{FOLLOW}_1(p_0, C)$
$\quad$ **for** $(n, C) \in \mathit{nStacks}$
$\quad$ **do** $ts \leftarrow \text{VALUE}(n)$
$\qquad \text{NEXT-LOOKAHEADS}_B(\mathit{la}, (ts, C, \text{GOTO}_0(ts, C)), \mathit{root}, n, \mathit{visited})$

**Figure 8:** Procedure $\text{NEXT-LOOKAHEADS}_B$.

- the compression scheme proposed in [Bigonha et al. 1983] (BCS);
- row displacement scheme (RDS), reported in [Dencker et al. 1984];
- significance distance scheme (SDS), proposed in [Beach 1974];
- row column scheme (RCS), proposed in [Tewarson 1967];
- supression of zeros scheme (SZS), reported in [Dencker et al. 1984]);
- graph coloring scheme (GCS), proposed in [Schmitt 1979];
- line elimination scheme (LES), proposed in [Bell, 1974];
- row merging scheme (RMS), reported in [Passos 2007].

Each method was tested using the grammars of C, C#, HTML, Java and VB programming languages. The result of this study is directly generalized to LALR($k_v$) parsers, due to the same layout of LALR(1) and LALR($k_v$) 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 propose infinite 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 $k_{max} = 2$ 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 $k_{max} \geq 3$, 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 conformance. The opposite situation occurs with Scheme and Oberon-2.

| Grammar | $k_{max} = 1$ | | $k_{max} = 2$ | | $k_{max} = 3$ | | $k_{max} = 4$ | |
|---|---|---|---|---|---|---|---|---|
| | Confs. | T. (ms) | Confs. | T. (ms) | Confs. | T. (ms) | Confs. | T. (ms) |
| Algol-60 | 61 | 670 | 61 | 2,480 | 61 | 4,837 | 61 | 18,017 |
| Scheme | 78 | 839 | 38 | 1,320 | 38 | 2,485 | 38 | 4,957 |
| Oberon-2 | 32 | 898 | 1 | 1,464 | 1 | 1,567 | 1 | 1,493 |
| Notus | 575 | 1,068 | 541 | 34,828 | 539 | 38,921 | 539 | 67,467 |

**Table 1:** Evaluation of increasing $k$ in automatic conflict removal.

Our tool was also used in building a parser for the Machĭna 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, $k_{max} = 2$ 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 of 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;
- 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, turning it viable in the sense that memory requirements are minimized.

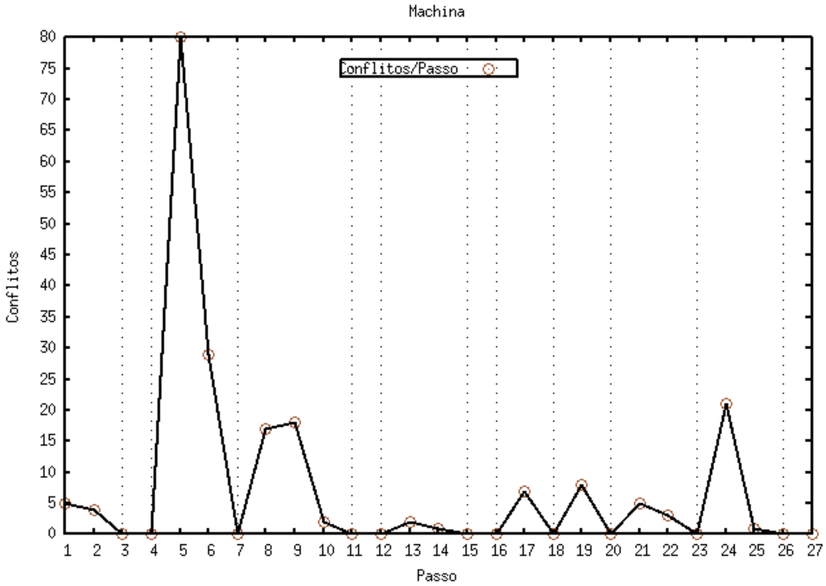


**Figure 9:** Conflict removal graph for the Machĭnaprogramming language.

All obtained results are based on empirical data, determined by using established and also new programming languages, such as Notus and Machĭna. 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

[Aho et al. 1986] Aho, A. V., Sethi, R., Ullman, J. D.: "Compilers, Principles, Techniques, and Tools"; Addison-Wesley, 1986.

[Beach 1974] Beach, B.: "Storage Organization for a Type of Sparce Matrix"; Proc. $5^{th}$ Southeastern Conference on Combinatorics, Graph Theory and Computing, 1974, 245-252.

[Bell, 1974] Bell, J. R.:"A Compression Method for Compiler Precedence Tables."; Proc. IFIP Congress, Elsevier North-Holland, New York (1974), 359–362.

[Bigonha et al. 1983] Bigonha, R. S. and Bigonha, M. A. S.:"A Method for Efficient Compactation of LALR(1) Parsing Tables"; Technical Report, Federal University of Minas Gerais, Programming Languages Laboratory (1983).

[Bigonha et al. 2007] Bigonha, R. S., Tirelo, F., Iorio, V. O., Bigonha, M. A. S.: "A Linguagem de Especificação Formal Machĭna 2.0"; Technical Report, Federal University of Minas Gerais, Programming Languages Laboratory (2007).

[Charles 1991] Charles, P.: "A Pratical Method for Constructing Efficient LALR($k$) Parsers with Automatic Error Recovery"; Doctoral Thesis, New York University (1991).

[Dencker et al. 1984] Dencker, P., Dürre, K., Heuft, H.: "Optimization of Parser Tables for Portable Compilers"; (TOPLAS) ACM Transactions on Programming Languages and Systems, 6, 4 (1984), 546-572.

[DeRemer and Pennello 1982] DeRemer, F., Pennello, F.: "Efficient Computation of LALR(1) Look-Ahead Sets"; (TOPLAS) ACM Transactions on Programming Languages and Systems, 4, 4 (1982), 615–649.

[Kristensen and Madsen 1981] Kristensen, B. B., Madsen, O. L.:"Methods for Computing LALR(k) Lookahead"; (TOPLAS) ACM Transactions on Programming Languages and Systems, 3, 1 (1981), 60–82.

[Passos et al. 2007] Passos, L. T., Bigonha, Mariza A. S., Bigonha, R. S.: "A Methodology for Removing LALR(k) Conflicts"; J.UCS (Journal for Universal Computer Science), 13, 6 (2007), 737–752.

[Passos 2007] Passos, L. T.: "Gerador LALR com Suporte a Resolução de Conflitos"; Master's Thesis, Federal University of Minas Gerais, Programming Languages Laboratory (2007).

[Schmitt 1979] Schmitt, A.: "Minimizing Storage Space of Sparse Matrices by Graph Coloring Algorithms"; Graphs, Data Structures, Algorithms, 1979, 157-168.

[Tewarson 1967] Tewarson, R. P.: "Row Column Permutation of Sparce Matrices"; Computer Journal; 10, 3 (1967), 300–305.

[Tirelo and Bigonha 2006] Tirelo, F., Bigonha, R.: "Notus"; Technical Report, Federal University of Minas Gerais, Programming Languages Laboratory (2006).

[Tomita 1991] Tomita, M.: "Generalized L.R. Parsing"; Kluwer Academic Publishers, Norwell (1991).