An LALR Parser Generator with Conflict Removal Support

Leonardo Passos[1], Mariza Bigonha[2], and Roberto Bigonha[2]

[1] Department of Computer Science, Federal University of the Jequitinhonha and Mucuri Valleys
leonardo.passos@ufvjm.edu.br
[2] Department of Computer Science, Federal University of Minas Gerais
{mariza, bigonha}@dcc.ufmg.br

Abstract. Context free grammars are used for numerous purposes, from the specification of concrete or abstract programming languages syntax to the definition of exchange formats in component-based software applications. To decrease the time spent in development time, the process of writing parsers from context free grammars is automated by the use of parser generator tools, which often restrict the input grammar to be LALR. However, due to the presence of conflicts and the constant absence of means to automatically remove them and high level facilities to elucidate their causes, writing an LALR grammar becomes a major bottleneck in software development. To address this problem, we present an LALR parser generator with automatic conflict removal, along with its similarities and differences when compared to other approaches, such as the ones implemented in SableCC and LPG. The proposed tool also implements a methodology to solve conflicts that cannot be automatically eliminated. In such case, high level conflict debug facilities, such as derivation trees, are supported. Furthermore, conflicts are listed according to an heuristic of removal; once a high level priority conflict is removed, all the others caused by the presence of the first may transparently be eliminated.

1 Introduction

Context free grammars are used for numerous purposes, from the specification of concrete or abstract programming language syntax to the de definition of exchange formats in component-based software applications [7]. To automate the process of parser construction, grammars are extensively used as input to parser generator tools, which output high performance non-backtracking parsers. Parser generators, however, often restrict input grammars to be LALR.

In case of non LALR grammars, parser generators report conflicts, which are nondeterministic points resulted from the grammar. Most of the times these conflicts have no clear indication of their cause if one inspects only the grammar. For that matter, LALR parser generators output log files containing all the internal details necessary to understand the cause of any given conflict. These

files, however, contain low level data, such as the parsing tables and the LALR automaton, and tend to be too extensive for practical use. To illustrate the difficulty of this approach, the removal of the 575 conflicts reported by the CUP parser generator [4] for the Notus programming language grammar [3] requires the simulation of the LALR automaton through its 332 states, dumped in a log file with 790 kb ($\approx$ 12,475 lines) of pure text data. Among the parser generators that we have investigated (Yacc [11], CUP [4], SableCC [10] and LPG [9]) only SableCC and LPG provide some level of automatic conflict removal. As for debugging, all tools fail to support any facility other than log files.

As an alternative to this scenario, we propose an LALR parser generator that supports automatic conflict removal for a subset of conflicts. The proposed tool also supports a methodology to solve conflicts that cannot be automatically eliminated. In such case, high level conflict debug facilities, such as derivation trees, are presented. Furthermore, conflicts are listed according to an heuristic of removal; once a high level priority conflict is removed, all the others caused by the presence of the first may transparently be eliminated.

This article is organized as follows: Section 2 discusses the sources of conflicts, followed by the explanation of an automatic conflict removal mechanism in Section 3. Next, Section 4 proposes a methodology for conflict removal. The adaptations required over the method outlined in Section 3 so as to support the methodology are presented in Section 5, along with the core implemented algorithms. Experimental results and related work are discussed in Sections 6 and 7. Section 8 concludes the article and discusses some future work.

2 Conflicts

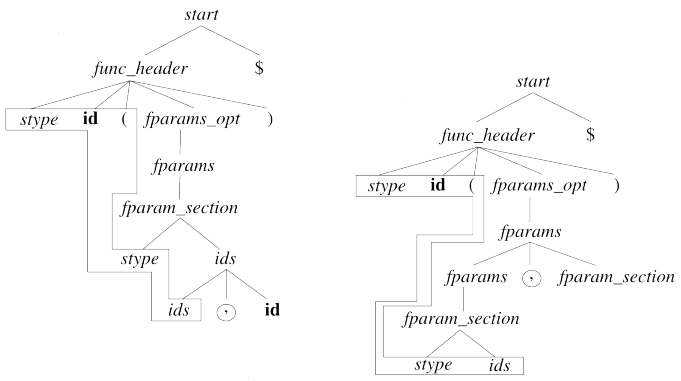
A conflict reported by an LALR parser generator indicates a non-deterministic point resulted from the grammar. Suppose, for example, the grammar in Figure 1 to express function headers. From this point on, the following convention is adopted: nonterminals are written in italics and terminals are typed in bold, delimited in double quotes or represented as single lower case letters.

func_header	$\rightarrow$ *stype* **id** "(" *fparams_opt* ")"	*fparam_section*	$\rightarrow$ *stype ids*
fparams_opt	$\rightarrow \lambda$	*stype*	$\rightarrow$ **float**
	| *fparams*		| **int**
fparams	$\rightarrow$ *fparam_section*	*ids*	$\rightarrow$ **id**
	| *fparams* "," *fparam_section*		| *ids* "," **id**

Fig. 1. Function header grammar.

The grammar in question is not LALR(1) due to a shift/reduce conflict under the "," token, as illustrated in Figure 2. The parser, if produced, having processed the

[3] The Notus programming language is a language designed in our laboratory to specify the denotational semantics of a programming language.



(a) First option: shift the "," token.

(b) Second option: reduce *stype ids* to *fparam_section*.

Fig. 2. Nondeterminism resulted from the grammar shown in Figure 1 when attempting to produce an LALR(1) parser. The sentential form processed by the parser is contained within a polygon. The lookahead token is delimited within a circle.

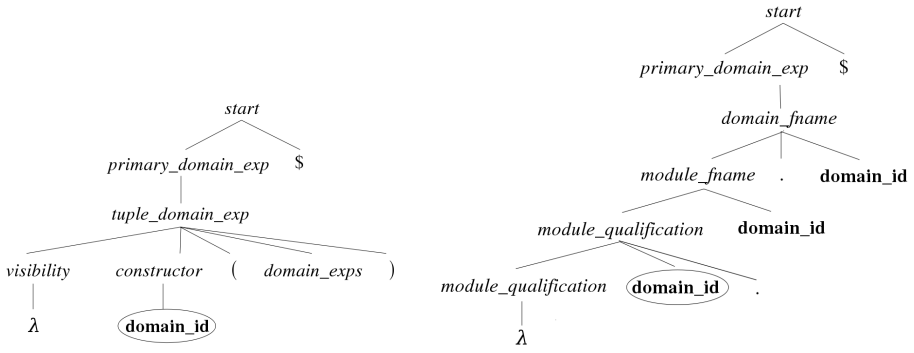
sentential form *stype* **id** *stype ids* faces two possibilities of execution: (a) consume "," and obtain the sentential form *stype* **id** *stype ids* ","; (b) substitute *stype ids* with *fparam_section*, i.e., reduce the sentential to *stype* **id** *fparam_section*.

The other type of conflict occurs when two or more reduce items in a state of the LALR automaton share the same lookahead. To illustrate such conflict, consider the grammar fragment from the Notus programming language shown in Figure 3. As illustrated by the derivation trees in Figure 4, the LALR(1) parser, having processed no sentential form, faces two possibilities under the **domain_id** lookahead : (a) reduce λ to the *visibility* nonterminal; (b) reduce λ to *module_qualification*.

primary_domain → *domain_fname* | *tuple_domain_exp*
primary_domain → *domain_fname* | *tuple_domain_exp*
domain_fname → *module_fname* "." **domain_id** | **domain_id**
module_fname → *module_qualification* **domain_id**
module_qualification → *module_qualification* **domain_id** "." | λ
tuple_domain_exp → *visibility constructor* "(" *domain_exps* ")"
...

Fig. 3. A fragment of the Notus programming language. The ... is used to omit irrelevant productions to our example.

Conflicts in non-LALR grammars are caused by lack of right context or ambiguity. The reduce/reduce conflict illustrated in Figure 4 is an example of an inappropriate amount of right context. If two lookaheads are used, however, the parser is able to decide which action to execute: (a) if the right con-



(a) First reduce option: reduce λ to *visibility*.

(b) Second reduce option: reduce λ to *module_qualification*.

Fig. 4. Nondeterminism resulted from the grammar shown in Figure 3 when attempting to produce an LALR(1) parser. The sentential form processed by the parser is contained within a polygon. The lookahead token is delimited within a circle.

text consists of the lookahead string **domain-id**".", then the parser reduces to *module-qualification*; (b) if the lookahead string is **domain-id**"(", then the parser reduces to *visibility*.

However, some conflicts caused by lack of right context may required an infinite amount of lookaheads. In these cases, the conflict is removed by rewriting some rules of the grammar without changing the language in question. Consider, for instance, the regular language $L = (b^+a) \cup (b^+b)$. A possible grammar G for L is:

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

For this grammar, an LALR(k) parser generator reports a reduce/reduce conflict involving the items $B_1 \rightarrow \mathbf{b.}$ and $B_2 \rightarrow \mathbf{b.}$, being b^k the conflict string. Increasing k does not remove the conflict; in fact, $\nexists k$ capable of it. This type of conflict can be removed if an equivalent LALR grammar G' is obtained by rewriting some grammar rules. In this case G' definitely exists, for L is a regular language. Nevertheless, it should be pointed out that this kind of solution is not always possible.

As for conflicts caused by ambiguity, consider an ambiguous grammar G to describe the syntax of expressions:

$$\begin{array}{l} exp \rightarrow exp\ \text{“+”}\ exp \mid exp\ \text{“-”}\ exp \\ exp \rightarrow exp\ \text{“*”}\ exp \mid exp\ \text{“/”}\ exp \\ exp \rightarrow \mathbf{id} \mid \mathbf{num} \mid \text{“(”}\ exp\ \text{“)”} \end{array}$$

It is well known that the corresponding language can be expressed by a non ambiguous LALR(1) set of rules, although is more probable that one will first write an ambiguous specification. The LALR(1) version of G is defined as [6]:

$$
\begin{array}{lll}
exp & \rightarrow exp \text{ “+” } term & \mid \; exp \text{ “-” } term \\
term & \rightarrow term \text{ “*” } factor & \mid \; term \text{ “/” } factor \\
factor & \rightarrow \textbf{id} \mid \textbf{num} \mid \text{“(” } exp \text{ “)”} &
\end{array}
$$

Ambiguity removal can also be achieved without rewriting the grammar, as it is the case of the use of adhoc solutions based on precedence and associativity [1]. Some ambiguity conflicts, however, simply cannot be removed from the grammar without altering the language in question. These conflicts are due to the existence of inherently ambiguous syntax constructions. An example is any set of rules that describe the language $\{a^m b^n c^k \mid m = n \vee n = k\}$ [6].

3 Automatic Conflict Removal

Some conflicts caused by lack of right context, as discussed in Section 2, can be automatically removed if the length of lookaheads is increased.

A possible solution is to produce LALR(k) parsers with the appropriate value of k. The difficulty of this approach is that the parsing tables substantially grows as k increases. The Action table, for instance, has lines indexed by the elements in M_0, the set of states of the LR(0) automaton, and columns represented by tokens of length k, where each token symbol belongs to an alphabet Σ. The number of entries is thus given by $|M_0| \times ((|\Sigma - \{\$\}|)^k + |\{\$\}|)$, where the bars denote the cardinality of the set being considered and \$ stands for the EOF marker. Table 1 demonstrates the Action table size according to k for five mainstream languages: C, C#, Java, HTML and Visual Basic .NET. Note that the C# full LALR(3) Action table requires more than two billion entries.

Grammar	$\mathbf{\lvert M_0 \rvert}$	$\mathbf{\lvert \Sigma \rvert}$	**k = 1**	**k = 2**	**k = 3**
C	352	85	29,920	2,484,064	208,632,160
C#	807	141	11,3787	15,818,007	2,214,408,807
HTML	348	129	44,892	57,019,80	729,809,244
Java	632	107	67,624	7,101,784	752,722,744
VB .Net	636	144	91,584	13,006,200	1,859,796,288

Table 1. LALR(k) Action table size as k increases.

Charles [3] proposes an alternative representation of LALR(k) parsers by taking k not as a fixed value, but as a variable limited by a constant k_{max}. To explain how this approach works, consider an entry Action$[q, a] = \{S2, R5, R6\}$, where $S2$ denotes a shift action to state 2 and $R5$ and $R6$ denote reduce actions by productions numbered 5 and 6, respectively. In the set of actions there are two shift/reduces ($\{S2, R5\}$, $\{S2, R6\}$) and one reduce/reduce conflict ($\{R5, R6\}$). Suppose the existence of a procedure $\mathcal{P}$ to simulate the execution of the LR(0) automaton with the goal of calculating the lookaheads that may follow a from state q if each parsing action is executed. Taking k as 3 results in the following possibilities: (a) if the shift action is performed, $\mathcal{P}$ determines that only the

lookaheads in $L_1 = \{bc, bd\}$ may follow a from state q; (b) if $\mathcal{P}$ reduces under production 5, then only the lookaheads in $L_2 = \{db, cc\}$ may follow a from a predecessor state of q, to which the parser would have gone under reduction; (c) if $\mathcal{P}$ reduces under production 6, then only the lookaheads in $L_3 = \{dc, da\}$ may follow a from a predecessor state of q, to which the parser would have gone under reduction. Putting these possibilities in the form of a tree results in the DFA depicted in Figure 5. The states q_1, q_2, q_3 and q_4 are not in M_0; they exist specifically for the purpose of the DFA. Since L_1, L_2 and L_3 are disjoint sets, each lookahead string in $\{a\}(L_1 \cup L_2 \cup L_3)$ uniquely identifies one parsing action to execute, as indicated in the DFA. Note, however, that from q_1 each lookahead token in the strings bc, bd and cc reaches a state whose out degree is one. Therefore, considering only the first symbol of such strings ($\{b, c\}$) is enough to determine which parsing action to execute. In contrast, the strings db, dc and da share the lookahead token d as prefix. Following it from q_1 leads to the state q_3, whose out degree is two. In this case, if d is found, the parser has more than one parse action to consider: $R5$ and $R6$. Another level of lookahead enables it to make such decision. The DFA can consequently be simplified by eliminating states q_2 and q_4. The length of lookahead strings starting with a, then varies from 2 to 3. This is exact varying characteristic of k: the produced parser can look at most k tokens ahead so as to decide which parsing action to execute. These parsers will be referred as LALR(k_v) parsers.

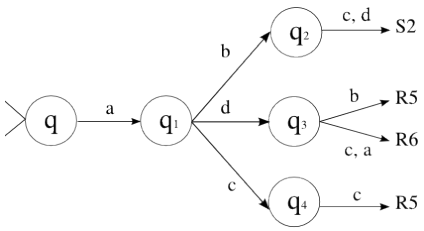


Fig. 5. DFA representing the lookahead string paths that may follow a from state q. In the DFA, each lookahead string leads to a unique parse action, which causes the conflict in Action$[q, a]$ to be eliminated.

The mechanism in which LALR parses operate can be altered in order to incorporate calculated lookahead DFAs, hereafter referred as LDFAs. For an entry Action$[q, a]$ with a conflict, the parser must deviate its execution to the corresponding calculated LDFA. To this purpose, Action$[q, a]$ must store a special kind of entry: a *lookahead* action to state q_1, the first state of the corresponding LDFA. Lookahead actions differ from shift actions in the sense that no token is consumed at any moment and LDFA states are not pushed onto the stack kept by the LALR driver program. Following each lookahead action from q_1, the LDFA continues its execution until a shift, reduce or error entry is reached. In this case, the parser performs the corresponding action and behaves as an ordi-

nary LALR parser. To facilitate transition from LALR states to LDFA states, each calculated LFDA has its transition function table appended at the end of the Action matrix. Altogether, LDFA transition tables comprise the *lookahead table*. To illustrate how this works, consider the augmented grammar in Figure 6.

$$
\begin{array}{lll}
S' \rightarrow S\ \mathbf{\$} & (0) & \qquad A_1 \rightarrow A_1\ \mathbf{a} \quad (3) \\
S \rightarrow A_1\ \mathbf{b}\ \mathbf{c} & (1) & \qquad\quad \mid\ \lambda \quad (4) \\
\quad \mid\ A_2\ \mathbf{b}\ \mathbf{d} & (2) & \qquad A_2 \rightarrow \mathbf{a} \quad (5)
\end{array}
$$

Fig. 6. A non LALR(1) grammar with a solvable shift/reduce conflict if 3 lookaheads are used.

This grammar results in a shift/reduce conflict $\{S1, R4\}$ in the first state of the LALR(1) automaton under the **a** terminal, as illustrated by the corresponding parsing tables in Figure 7. Using k_{max} as 3, the process $\mathcal{P}$ that simulates execution of the LR(0) automaton faces two possibilities: (i) execute $S1$: $\mathcal{P}$ then goes to state 1 under the **a** token. In state 1, it performs a reduce under $A_2 \rightarrow \mathbf{a}.$, which brings it back to state 0 with the transition symbol A_2. From 0 under A_2, $\mathcal{P}$ reaches state 2. From 2, the only possible transition occurs under the **b** terminal, which then leads $\mathcal{P}$ to state 8. Again, the only possible action to be executed is a shift under **d** to state 9. Since k has now reached 3, $\mathcal{P}$ determined that by executing the shift action the only lookahead string that may follow a is bd; (ii) execute $R4$: $\mathcal{P}$ reduces under $A_1 \rightarrow \lambda$, which brings it back to state 0. Following A_1 from 0 leads $\mathcal{P}$ to state 3, in which two possibilities arise: (a) shift **b** and go to state 5; (b) shift **a** and go to state 6. Simulating (a), $\mathcal{P}$ goes to state 5, whose only possible action is a shift under **c** to state 7. From 7, simulation does not go any further because the calculated lookahead string bc, when appended to a, has length equal to k_{max}. Following the execution path indicated by (b), $\mathcal{P}$ reaches state 6. At this point, the only possible action to perform is a reduce under $A_1 \rightarrow A_1\mathbf{a}$, which then brings $\mathcal{P}$ back to state 0. From 0, it again follows a transition to state 3 under the A_1 nonterminal. This process continues until the lookahead string aaa is found, which causes simulation to stop.
From this, the conflict $\{S1, R4\}$ can be removed from Action$[0, a]$: under lookaheads aaa and abc the parser reduces under the 4^{th} production; under abd the parser goes to state 1. From the conflict symbol a, the lookahead string aa shares no prefix with bc and bd. Therefore, the extra level of lookahead inspection in aaa is useless; having inspected only aa is enough to identify that $S1$ is to be executed. The new Action table, appended with the lookahead table, is given in Figure 8.

The discussed example shows the intuition of how LALR(k_v) are generated using Charles approach. It is worth mentioning that simulation of the LR(0) automaton as performed may enter infinite loop if the input grammar G is circular, i.e., given a nonterminal A, A derives A in one or more steps, or G results in an LR(0) automaton with cycles whose transition symbols are all

	Action Table					*Goto Table*		
	\$	a	b	c	d	S	A_1	A_2
0		$S1, R4$	$R4$			4	3	2
1			$R5$					
2			$S8$					
3		$S6$	$S5$					
4	*accept*							
5				$S7$				
6		$R3$	$R3$					
7	$R1$							
8					$S9$			
9	$R2$							

Fig. 7. Parsing tables for the grammar in Figure 6.

nullable nonterminals. In case of such grammars, hereafter referred as *NLALR grammars*, the calculation of LDFAs is stopped. If continuation is performed, Charles states that termination will never be reached. Stopping the calculation of LDFAs directly impacts the methodology supported by our parser generator, as it is discussed in Section 4.

	Action Table				
	\$	a	b	c	d
0		$L10$	$R4$		
...	...				
	Lookahead Table				
	\$	a	b	c	d
10		$R4$	$L11$		
11				$R4$	$S1$

Fig. 8. Parsing tables for the grammar in Figure 6. The dots omit the states that are the same as in Figure 7.

4 A Methodology for Conflict Removal

The proposed parser generator is embedded in a graphical user interface so as to facilitate the support of a methodology for conflict removal. Figure 9 presents a screenshot of this GUI, called SAIDE (Syntax Analyzer Integrated Development Environment). 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. This 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 possibly 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 9, 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. 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 9. For each conflict, the user can also inspect the set of lookaheads used as an attempt to remove the reported conflict.

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 consulting 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.

5 Implemented Algorithms

In this section we discuss the implemented algorithms that support the methodology presented in Section 4. Due to the lack of space, we do not present the algorithms used in the calculation of derivation trees as means to elucidate conflicts. For that matter, the reader is referred to the paper by DeRemer and Pennelo on lookaheads calculation [5], which presents such algorithms.

5.1 Automatic Conflict Removal

Charles calculates the lookaheads necessary to extend a given token by simulating the steps of the LR(0) automaton. When faced with NLALR grammars, the process of calculating LDFAs is 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 the user a wrong impression of the total amount of conflicts, disables SAIDE's capacity to determine the order

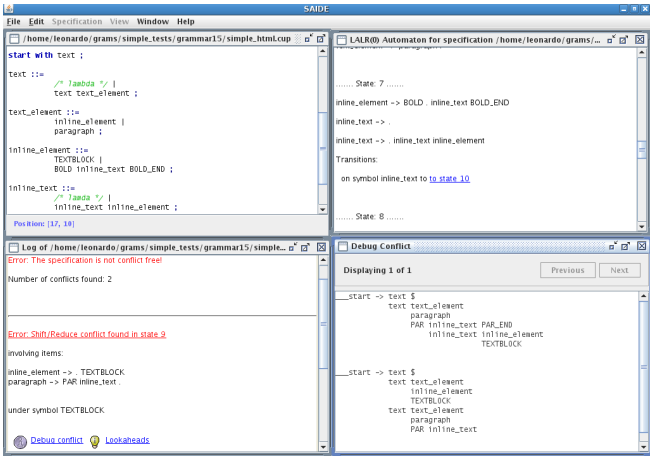


Fig. 9. SAIDE's main window.

in which conflicts should be removed and present the calculated lookaheads used as an attempt to remove a given conflict. In this manner, SAIDE's methodology as originally proposed becomes inapplicable[4].

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 originally four proposed by Charles.

The whole process starts with SWEEP, presented in Figure 10. Given a state p, for each entry Action$[p, a]$ that has at least two actions, SWEEP calls the FOLLOW-SOURCES$_A$ procedure. This procedure, shown in Figure 11, is a façade procedure to FOLLOW-SOURCES$_B$, presented in Figure 12. Given an initial stack as kept by the LALR driver program, FOLLOW-SOURCES$_B$ simulates the execution of an action in Action$[p, a]$ in order to define the set of stacks that result in the reading of a. SWEEP then associates each set of obtained stacks with the corresponding action by means of a mapping structure called *sources*. For a shift action s in Action$[p, a]$, SWEEP makes $sources(s) = \{([p], a)\}$, where $[p]$ denotes a stack with state p as its top and only element and a as the string obtained so far. In case of a reduce action r by a production $A \rightarrow \alpha$ in Action$[p, a]$, FOLLOW-SOURCES$_B$ is called for each possible start stack $[p_0]$, where p_0 is a predecessor state of p under the α sentential form in the LR(0) automaton, as returned by the PRED function. FOLLOW-SOURCES$_B$ then continues simulation by making a transition from p_0 to the state $q = \text{GOTO}_0[p_0, A]$, which results in a new stack $[p_0 q]$. GOTO$_0$ stands as the transition function of the LR(0) automaton, defined over $M_0 \times \textit{Grammar Symbol} \rightarrow M_0$. A simulation path at this stage stops as soon as a appears as a reading token. While calculating sources(r), if FOLLOW-SOURCES$_B$, given a current stack and a transition symbol X, reaches a state s

[4] For a further discussion, please refer to http://sourceforge.net/projects/lpg/forums/forum/523519/topic/3413786.

with an item of the form $C \to \gamma \,.\, X$, it checks two situations: (i) if the current stack has at least $|\gamma| + 1$ states, then popping $|\gamma|$ states makes the predecessor state of s under γ the top state of the stack; (ii) otherwise, the current stack has less than $|\gamma| + 1$ states; popping $|\gamma|$ states would cause underflow. In such case, let $\gamma = \gamma_1\gamma_2$, where $|\gamma_2|+1$ is equal to the size of the stack. If p_0 is a predecessor state in the bottom of the stack under γ_1, reduction is then simulated by creating a new stack $[p_0]$ and following a transition to $\text{GOTO}_0(p_0, C)$, which results in a recursive call to FOLLOW-SOURCES$_B$.

```
SWEEP(p)
 1  for a ∈ Σ
 2  do if |Action[p, a]| > 0 ∧ a ≠ $
 3        then sources ← ∅
 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)
```

Fig. 10. Main procedure to automatically remove conflicts.

```
FOLLOW-SOURCES_A(sources, stack, X, a, w)
1  ASSERT(stack = [p1...pn])
2  root ← node ← NODE((p1, λ))
3  for 2 ≤ i ≤ n
4  do node2 ← NODE((pi, λ))
5     ADD-CHILD(node, node2)
6     node ← node2
7  SND(VALUE(node)) ← w
8  FOLLOW-SOURCES_B(sources, (pn, X, GOTO0(pn, X)), a, w, root, node, ∅, ∅)
```

Fig. 11. Façade procedure FOLLOW-SOURCES$_A$.

The first difference when compared with Charles proposal is the presence of two FOLLOW-SOURCES procedures. 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 [8] and it is used by FOLLOW-SOURCES$_B$ as a means 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

```
FOLLOW-SOURCES_B(sources, transition, a, w, root, node, visited, roots)
 1  stackSize ← COUNT(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 (node_2 ← PARENT(node_2)) ≠ nil
20                    do list ← list + [FST(VALUE(node_2))]
21
22                    ASSERT(list = [p_n ... p_1])
23                    stack ← [p_1 ... p_n q]
24                    sources ← sources ∪ {(stack, wa)}
25  bottom ← FST(VALUE(root))
26  for C → γ• ∈ ts | C ≠ 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 = γ_1γ_2), where |γ_2| = stackSize − 1
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)
```

Fig. 12. Procedure FOLLOW-SOURCES$_B$.

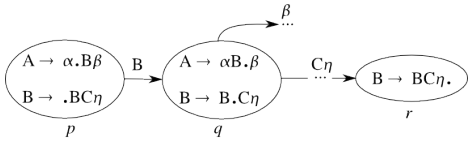


Fig. 13. Subgraph of a LR(0) automaton with a parsing cycle. Reproduced from [3].

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 the NODE constructor 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$_B$ requires eight arguments: 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 whose components are a source state, a transition-symbol and a destination-state), the terminal a, the current processed string w, that corresponds to λ when FOLLOW-SOURCES$_A$ is first called in SWEEP, 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* and wa is the lookahead string obtained from the simulation process that resulted in *stack*. The first part of loop control in FOLLOW-SOURCES$_B$ is located in lines $2-8$. Although also present in Charles algorithm, the presented code is an adaptation so as to work with the tree structure and the absence of assumptions related to the input grammar and the corresponding LR(0) automaton. Charles loop control guarantees that no transition is revisited from any given state and transition symbol. In our case we must guarantee that only one execution of FOLLOW-SOURCES$_B$ occurs for a given stack and transition symbol. This prevents infinite loops like the one illustrated in Figure 13. If simulation reaches a state p, follows a transition under a nullable nonterminal B to state q, and from it produces $C\eta$, where $C\eta \stackrel{*}{\Rightarrow} \lambda$, simulation would then come back to state p, where this process could continue indefinitely.

Loop control is performed each each time FOLLOW-SOURCES$_B$ is activated. 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 COUNT, which returns the number of nodes in the path 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, *visited* is used. Lines $10-24$ 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*. Another loop control, not performed in Charles' algorithm, occurs in

```
RESOLVE-CONFLICTS(q, t, sources, n)
 1  if t = $ ∨ n > k_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)
```

Fig. 14. Procedure RESOLVE-CONFLICTS.

the 10^{th} line. If following a transition from $q = \text{GOTO}_0(ts, X)$, where ts stands as the top state in the current stack, under a nullable nonterminal Y results in an instant configuration $([p_1p_2...p_nqq_1q_2...q_nq], w)$ obtained in one or more steps from a previous configuration $([p_1p_2...p_nq], w)$, then there is a cycle over nullable nonterminals in the LR(0) automaton. Detecting cycles using the tree structure is straight forward: given a *node* and a value (q, w), a cycle occurs if GET-FROM finds a node in the path from *node* (inclusively) to the root of the tree stack whose value coincide with (q, w). Lines $26 - 37$ are responsible for making reductions. As mentioned before, reductions while simulating the LR(0) automaton may cause underflow. 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 γ_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 14, checks if a conflict is removed. If not, it generates 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 either case, the extension of lookaheads does not go further. 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 set in Action$[p, a]$. 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).

```
NEXT-LOOKAHEADS_A(stack, t)
1  la ← ∅
2  ASSERT(stack = [p_1...p_n])
3  root ← node ← NODE(p_1)
4  for 2 ≤ i ≤ n
5  do node_2 ← NODE(p_i)
6      ADD-CHILD(node, node_2)
7      node ← node_2
8  NEXT-LOOKAHEADS_B(la, (p_n, t, GOTO_0(p_n, t)), root, node, ∅)
9  return la
```

Fig. 15. Façade procedure NEXT-LOOKAHEADS$_A$.

Analogous to FOLLOW-SOURCES$_A$, NEXT-LOOKAHEADS$_A$, shown in Figure 15, is a façade procedure to NEXT-LOOKAHEADS$_B$. It structures the received stack in a tree format.

Having such tree, NEXT-LOOKAHEADS$_B$, presented in Figure 16, fast calculates the tokens that can be found given a stack and a transition symbol. To achieve this, it uses two external functions defined in [5]: READ$_1$ and FOLLOW$_1$. 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 $C \rightarrow \alpha \bullet X\beta \in p$, $\beta \stackrel{*}{\Rightarrow} \lambda$ and $p_0 \in$ PRED(p, α). From a received stack tree and transition, NEXT-LOOKAHEADS$_B$ first grabs all tokens returned by READ$_1$ for the given transition. Reductions are treated as in FOLLOW-SOURCES$_B$, except that in cases of underflow, simulation does not proceed. 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, not present in Charles algorithm, is achieved just like FOLLOW-SOURCES$_B$, i.e., by storing all visited stacks.

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

```
NEXT-LOOKAHEADS_B(la, transition, root, node, visited)
 1  ASSERT(transition = (ts, X, q))
 2  bottom ← VALUE(root)
 3  if (ID(node), transition) ∈ visited
 4    then return
 5  la ← la ∪ READ_1(ts, X)
 6  stackSize ← COUNT(node, root)
 7  nStacks ← ∅
 8  for C → γ • Xδ | δ ⇒* λ ∧ C ≠ S
 9  do if |γ| + 1 < stackSize
10        then node_2 ← UP(node, |γ|)
11             nStacks ← nStacks ∪ {(node_2, C)}
12
13        else ASSERT(γ = γ_1γ_2), where |γ_2| = stackSize − 1
14             for p_0 ∈ PRED(bottom, γ_1)
15             do la ← la ∪ FOLLOW_1(p_0, C)
16  for (n, C) ∈ nStacks
17  do ts ← VALUE(n)
18     NEXT-LOOKAHEADS_B(la, (ts, C, GOTO_0(ts, C)), root, n, visited)
```

Fig. 16. Procedure NEXT-LOOKAHEADS$_B$.

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 entries with more than one parse action 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 LDFA, and thus obtain the strings of length up to k_{max} for which the conflict

is not removed. The number of conflicts for Action$[p, a]$ is given by $(|shift| \times |reds|) + (\lambda x.\ if\ x \geq 2\ then\ 1\ else\ 0)|reds|$, where *shift* stands as the set of shift actions and *reds* as the set of reduce actions in *acts*. For the conflicts that could not be automatically removed, SAIDE lists them in a heuristic manner so as to prioritize the order in which they should be removed. 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. If the user follows the heuristic listing, conflicts with higher priority, when removed, may cause the transparent elimination of other conflicts listed after them due to the flow of lookahead propagation indicated by the conflict graph.

6 Related Work

SableCC [10] is an LALR(1) parser generator with automatic conflict removal support. SableCC attempts to automatically remove conflicts by inlining productions. To illustrate how this process works, consider the following grammar G:

$$S \rightarrow A_1\ \mathbf{b}\ \mathbf{c}\ \mid\ A_2\ \mathbf{b}\ \mathbf{d} \qquad \begin{array}{l} A_1 \rightarrow \mathbf{a}\ \mid\ \mathbf{e} \\ A_2 \rightarrow \mathbf{a}\ \mid\ \mathbf{f} \end{array}$$

This grammar is not LALR(1) due to a reduce/reduce conflict involving productions $A_1 \rightarrow a$ and $A_2 \rightarrow a$. Since b can follow A_1 and A_2, having processed a, the parser has two possibilities of reduction. The inline strategy is to postpone reductions, so as to increase the parser's right context. For each production P whose right hand side contains either A_1 or A_2, P is redefined so that each occurrence of A_1 and A_2 is substituted by their corresponding right hand sides. The nonterminals A_1 and A_2 are then removed from the grammar, along with their productions. This process results in a grammar G' equivalent to G, but LALR(1):

$$S \rightarrow \mathbf{a\ b\ c}\ \mid\ \mathbf{a\ b\ d}\ \mid\ \mathbf{e\ b\ c}\ \mid\ \mathbf{e\ b\ d}\ \mid\ \mathbf{f\ b\ c}\ \mid\ \mathbf{f\ b\ d}$$

When submitting an input grammar to SableCC, transformations do not change G explicitly; they are internal and transparent to users. SableCC does not handle recursive productions, either direct or indirect, so as to prevent infinite execution of the inlining process. If the left hand side of a production that participates in a conflict is recursive, or the conflict itself cannot be eliminated by inling productions, SableCC stops execution and reports the current conflict.

For each processed grammar, SableCC lists at most one conflict at a time. The discussed technique is less powerful when compared to the generation of LDFAs in the sense that it only handles non-recursive productions.

LPG [9] is another LALR parser generator with automatic conflict support. LPG is a continuation project over JikesPG, the LALR parser generator from IBM that implements the algorithms proposed by Philip Charles in [3]. LPG inherents the same problem from JikesPG in which SAIDE solves: the incorrect feedback of the number of conflicts when dealing with NLALR grammars. To illustrate this, consider the following grammar:

$$\begin{array}{ll} A \rightarrow B\ C\ D & C \rightarrow \mathbf{c}\ \mid\ \lambda \\ A \rightarrow B\ C\ E\ \mid\ \lambda & D \rightarrow \mathbf{d}\ \mid\ \lambda \\ & E \rightarrow \mathbf{e}\ \mid\ \lambda \end{array}$$

This grammar is not LALR for any k. Using k as one, LPG reports 3 shift/reduces and 1 reduce/reduce conflict. If executed using 2 lookaheads, LPG reports only the reduce/reduce conflict. It also reports that states 1, 2, 3 and 4 of the corresponding automaton are cyclic and that $B \stackrel{+}{\Rightarrow} B$. At this point, the user may have the false understanding that all shift/reduces were removed using an extra level of lookaheads, while the reduce/reduce conflict could not be eliminated due to the cyclic characteristic of states 1, 2, 3 and 4 and/or that B derives itself in one or more steps!

As for debugging facilities SableCC and LPG fail to provide any mechanism other than log messages. In fact, the reported log messages are not as complete as in other parser generators such as CUP, Bison and Yacc: SableCC does not dump the LALR automaton; LPG only print states of the LALR automaton that contain conflicts.

7 Experiments

We performed tests in order to evaluate the automatic conflict mechanism when applied to 3 input grammars: Algol-$60_{(1)}$, Algol-$60_{(2)}$ and Notus. Algol-$60_{(1)}$ is a grammar that supports a subset of the full Algol-60 programming language whereas Algol-$60_{(2)}$ is a complete specification[5]. The obtained results were:

Grammar	$\mathbf{k}_{max} = \mathbf{1}$		$\mathbf{k}_{max} = \mathbf{2}$		$\mathbf{k}_{max} = \mathbf{3}$	
	Confs.	T. (ms)	*Confs.*	T. (ms)	*Confs.*	T. (ms)
Algol-$60_{(1)}$	61	397.11	61	1,550.2	61	3,089.75
Algol-$60_{(2)}$	255	421.87	245	3,499.68	243	14,773.74
Notus	575	401.52	541	15,375.2	539	17,168.96

where *Confs* stands as the number of reported conflicts and T denotes execution time, measured in milliseconds. In Algol-60_1 the number of conflicts is not affected at any time whereas in Algol-60_2 there is a reduction of 4% and 5% using two and three lookaheads respectively. From the table, one concludes that

[5] These grammars are available in www.dcc.ufmg.br/~leonardo/saide/grammars

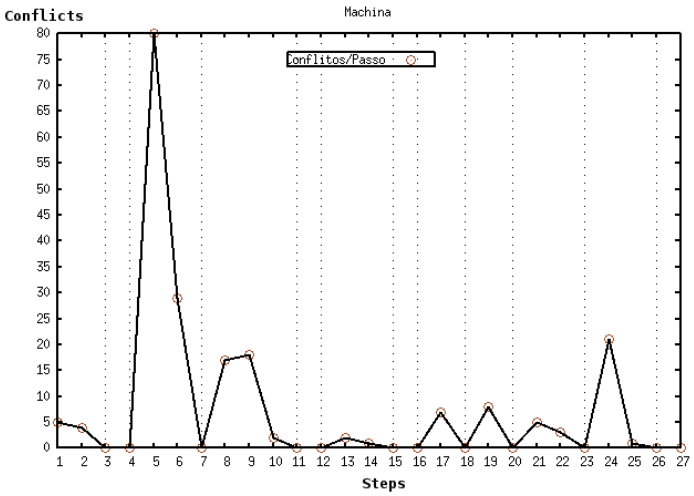


Fig. 17. Conflict removal graph for the Machĭnaprogramming language.

the productions omitted in Algol-60_1 are responsible for most of the conflicts in Algol-60_2. These conflicts result in a difference of execution time, specially for $k \geq 2$. For the Notus grammar, there is a reduction in 6% when using at most two lookaheads. Due to its complexity and extensiveness, the Notus grammar requires the highest amount of time when attempting to remove its conflicts.

In practice, SAIDE is fast when used for parsers that are built incrementally, with the adding of syntax constructions only when the number of conflicts in the current increment becomes zero. We believe that this is the most common scenario in parser development. In such cases, there is generally a small amount of conflicts to be tried to be automatically removed at each time, which causes a very small impact on execution time.

As a case study so as to evaluate the proposed methodology, the tool was used in building a parser for the Machĭna programming language[6]. The writing of its syntax specification was incrementally performed. 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 17 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.

[6] Machĭna is a programming language designed at our laboratory used in the specification of operational semantics of programming languages.

8 Conclusion and Future Work

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; Charles' algorithms were modified in order to accept any context free grammar. This makes the methodology independent of grammar characteristics.

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.

As for future work, we are currently applying compression techniques to the produced LALR(k_v) parsing tables. Preliminary results show that the generated parsing tables can be compressed in more than 90%, a result similar to LALR(1) compression rates.

We are also documenting the implemented code and integrating it to the Netbeans platform so as to make the tool publicly available. The official project web page for SAIDE can be found at http://www.dcc.ufmg.br/~leonardo/saide.

References

1. Aho, A. V. and Johnson, S. C. and Ullman, J. D.: Deterministic Parsing of Ambiguous Grammars, POPL '73: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages 1–21, 1973.
2. Bison: The GNU Parser Generator. In: http://www.gnu.org/software/bison/.
3. Charles, Philip: A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery, PhD thesis, New York University, 1991.
4. CUP Parser Generator. In: http://www2.cs.tum.edu/projects/cup/.
5. DeRemer, Frank and Pennello, Thomas: Efficient Computation of LALR(1) Look-Ahead Sets, ACM Transactions on Programming Languages and Systems, 4(4):615–649, 1982.
6. Aho, Alfred V. and Lam, Monica S. and Sethi, Ravi and Ullman, Jeffrey D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing, 2006.
7. Klint, Paul and Lämmel, Ralf and Verhoef, Chris.: Toward an Engineering Discipline for Grammarware, ACM Transactions on Software Engineering Methodology, 14(3):331–380, 2005.
8. Kristensen, Bent Bruun and Madsen, Ole Lehrmann: Methods for Computing LALR(k) Lookahead, ACM Transactions on Programming Languages and Systems, 3(1):60–82, 1981.
9. LALR Parser Generator. In: http://sourceforge.net/projects/lpg/
10. SableCC Parser Generator. In: http://sablecc.org/.
11. Johnson, S.C.: Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.