On LR Parsing with Selective Delays

Eberhard Bertsch[1], Mark-Jan Nederhof[2,⋆], and Sylvain Schmitz[3]

[1] Ruhr University, Faculty of Mathematics, Bochum, Germany
[2] School of Computer Science, University of St. Andrews, UK
[3] LSV, ENS Cachan & CNRS, Cachan, France

Abstract. The paper investigates an extension of LR parsing that allows the delay of parsing decisions until a sufficient amount of context has been processed. We provide two characterizations for the resulting class of grammars, one based on grammar transformations, the other on the direct construction of a parser. We also report on experiments with a grammar collection.

1 Introduction

From a grammar engineer's standpoint, LR parsing techniques, like the LALR(1) parsers generated by yacc or GNU/bison, suffer from the troublesome existence of *conflicts*, which appear sooner or later in any grammar development. Tracing the source of such conflicts and refactoring the grammar to solve them is a difficult task, for which we refer the reader to the accounts of Malloy et al. [15] on the development of a C# grammar, and of Gosling et al. [10] on that of the official Java grammar.

In the literature, different ways have been considered to solve conflicts automatically while maintaining a *deterministic* parsing algorithm—which, besides efficiency considerations, also has the considerable virtue of ruling out ambiguities—, such as unbounded regular lookaheads [6], noncanonical parsers [25], and delays before reductions [14]. Bertsch and Nederhof [4] have made a rather counter-intuitive observation on the latter technique: increasing delays uniformly throughout the grammar can in some cases introduce new conflicts.

In this paper we propose a parsing technique that *selects* how long a reduction must be delayed depending on the context. More interestingly, and unlike many techniques that extend LR parsing, we provide a *characterization*, using grammar transformations, of the class of grammars that can be parsed in a LR fashion with selective delays. More precisely,

– we motivate in Section 2 the interest of ML(k, m) parsing on an exerpt of the C++ grammar, before stating the first main contribution of the paper: we reformulate the technique of Bertsch and Nederhof [4] as a grammar transformation, and show how selective delays can capture non-ML(k, m) grammars,

⋆ Part of this research was conducted while the second author was an invited professor visiting ENS Cachan.

– we define the class selML(k, m) accordingly through a nondeterministic grammar transformation, which allows us to investigate its properties (Section 3),
– in Section 4 we propose an algorithm to generate parsers with selective delays, and prove that it defines the same class of grammars.
– We implemented a Java proof of concept for this algorithm (see `http://www.cs.st-andrews.ac.uk/~mjn/code/mlparsing/`), and report in Section 5 on the empirical value of selective delays, by applying the parser on a test suite of small unambiguous grammars [2, 22].
– We conclude with a discussion of related work, in Section 6.

Technical details can be found in the full version of this paper at `http://hal.archives-ouvertes.fr/hal-00769668`.

Preliminaries. We assume the reader to be familiar with LR parsing, but nonetheless recall some definitions and standard notation.

A *context-free grammar* (CFG) is a tuple $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ where N is a finite set of *nonterminal* symbols, Σ a finite set of *terminal* symbols with $N \cap \Sigma = \emptyset$—together they define the *vocabulary* $V = N \uplus \Sigma$—, $P \subseteq N \times V^*$ is a finite set of *productions* written as rewrite rules "$A \to \alpha$", and $S \in N$ the *start* symbol. The associated *derivation* relation $\Rightarrow$ over V^* is defined as $\Rightarrow = \{(\delta A\gamma, \delta\alpha\gamma) \mid A \to \alpha \in P\}$; a derivation is *rightmost*, denoted $\Rightarrow_{\mathrm{rm}}$, if γ is restricted to be in Σ^* in the above definition. The *language* of a CFG is $L(\mathcal{G}) = \{w \in \Sigma^* \mid S \Rightarrow^* w\} = \{w \in \Sigma^* \mid S \Rightarrow^*_{\mathrm{rm}} w\}$.

We employ the usual conventions for symbols: nonterminals in N are denoted by the first few upper-case Latin letters A, B, ..., terminals in Σ by the first few lower-case Latin letters a, b, ..., symbols in V by the last few upper-case Latin letters X, Y, Z, sequences of terminals in Σ^* by the last few lower-case Latin letters u, v, w, ..., and mixed sequences in V^* by Greek letters α, β, etc. The empty string is denoted by ε.

Given $\mathcal{G} = \langle N, \Sigma, P, S\rangle$, its *$k$-extension* is the grammar $\langle N \uplus \{S^\dagger\}, \Sigma \uplus \{\#\}, P \cup \{S^\dagger \to S\#^k\}, S^\dagger\rangle$ where # is a fresh symbol. A grammar is *LR(m)* [13, 23] if it is reduced—i.e. every nonterminal is both accessible and productive—and the following *conflict* situation does not arise in its m-extension:

$$\begin{aligned} &S^\dagger \Rightarrow^*_{\mathrm{rm}} \delta A u \Rightarrow_{\mathrm{rm}} \delta\alpha u = \gamma u && \delta \neq \delta' \text{ or } A \neq B \text{ or } \alpha \neq \beta \\ &S^\dagger \Rightarrow^*_{\mathrm{rm}} \delta' B v \Rightarrow_{\mathrm{rm}} \delta'\beta v = \gamma w v && m : u = m : wv \end{aligned}$$

where "$m : u$" denotes the prefix of length m of u, or the whole of u if $|u| \leq m$.

2 Marcus-Leermakers Parsing

The starting point of this paper is the formalization proposed by Leermakers [14] of a parsing technique due to Marcus [16], which tries to imitate the way humans parse natural language sentences. Bertsch and Nederhof [4] have given another, equivalent, formulation, and dubbed it "ML" for Marcus-Leermakers.

The idea of uniform ML parsing is that all the reductions are delayed to take place after the recognition of a fixed number k of *right context* symbols, which can contain nonterminal symbols. Bertsch and Nederhof [4] expanded this class by considering m further symbols of terminal lookahead, thereby defining ML(k, m) grammars. In Section 2.2, we provide yet another view on uniform ML(k, m) grammars, before motivating the use of selective delays in Section 2.3. Let us start with a concrete example taken from the C++ grammar from the 1998 standard [11].

2.1 C++ Qualified Identifiers

First designed as a preprocessor for C, the C++ language has evolved into a complex standard. Its rather high level of syntactic ambiguity calls for nondeterministic parsing methods, and therefore the published grammar makes no attempt to fit in the LALR(1) class.

We are interested in one particular issue with the syntax of *identifier expressions*, which describe a full name specifier and identifier, possibly instantiating template variables; for instance, "`A::B<C::D>::E`" denotes an identifier "`E`" with name specifier "`A::B<C::D>`", where the template argument of "`B`" is "`D`" with specifier "`C`".

The syntax of identifier expressions is given in the official C++ grammar by the following (simplified) grammar rules:

$$I \to U \mid Q, \quad U \to i \mid T, \quad Q \to N\,U, \quad N \to U \mathrel{::} N \mid U \mathrel{::}, \quad T \to i\,\texttt{<}I\texttt{>}.$$

An identifier expression I can derive either an *unqualified identifier* through nonterminal U, or a *qualified identifier* through Q, which is qualified through a *nested name specifier* derived from nonterminal N, i.e. through a sequence of unqualified identifiers separated by double colons "`::`", before the identifier i itself. Moreover, each unqualified identifier can be a *template identifier* T, where the *template argument*, between angle brackets "`<`" and "`>`", can again be any identifier expression.

Example 1. A shift/reduce conflict appears with this set of rules. A parser fed with "`A::`", and seeing an identifier "`B`" in its lookahead window, has a nondeterministic choice between

– *reducing* "`A::`" to a single N, in the hope that "`B`" will be the identifier qualified by "`A::`", as in "`A::B<C::D>`", and
– *shifting* the identifier, in the hope that "`B`" will be a specifier of the identifier actually qualified, for instance "`E`" in "`A::B<C::D>::E`".

An informed decision requires an exploration of the specifier starting with "`B`" in search of a double colon symbol. The need for unbounded lookahead occurs if "`B`" is the start of an arbitrarily long template identifier: this grammar is not LR(k) for any finite k.

Note that the double colon token might also appear inside a template argument. Considering that the conflict could also arise there, as after reading

"`A<B::`" in "`A<B::C<D::E>::F>::G`", we see that it can be arduous to know whether a "::" symbol is significant for the resolution of the conflict or not. In fact, this is an example of a conflict that *cannot* be solved by using regular lookahead as proposed in [5, 3, 8], because keeping track of the nesting level of well-balanced brackets is beyond the power of regular languages.[1]

2.2 Uniform ML

Observe that, in our extract of the C++ grammar, if we were to *postpone* the choice between the two possible actions and attempt to parse an N in full, then the issue would disappear. The mechanism Leermakers [14] employs for delaying parsing decisions is to extend a nonterminal with additional terminal and nonterminal symbols from its right context, thus delaying reduction to that nonterminal until the moment when these additional symbols have been parsed in full. This also involves introducing a new end-of-file terminal "#".

We refer the reader to [14, 4] and the full version of this paper for the details of the ML(k, m) parser construction. The automaton obtained by applying this construction on the C++ grammar is too large to be rendered on a single page. In what follows we present an alternative characterization of ML parsing on the basis of a grammar transformation.

Uniform ML as a Transformation. Although Leermakers does not present his technique in these terms, the intuition of extending nonterminals with right context can be realized by a grammar transformation that introduces nonterminals of the form $[A\delta]$ in $N' = N \cdot V^{\leq k}$, which combine a nonterminal A with its immediate right context δ.

This results for $k = 1$ and our C++ example into an LALR(1) grammar with rules:

$$\begin{gathered}
[I\#] \to [U\#] \mid [Q\#], \qquad [I\texttt{>}] \to [U\texttt{>}] \mid [Q\texttt{>}], \\
[U\#] \to i\,\# \mid [T\#], \quad [U\texttt{>}] \to i\,\texttt{>} \mid [T\texttt{>}], \quad [U\,\texttt{::}] \to i\;\texttt{::} \mid [T\,\texttt{::}], \quad [U] \to i \mid [T], \\
[Q\#] \to [NU]\,\#, \qquad [Q\texttt{>}] \to [NU]\,\texttt{>}, \\
[NU] \to [U\,\texttt{::}]\,[NU] \mid [U\,\texttt{::}]\,[U], \\
[T\#] \to i\,\texttt{<}\,[I\texttt{>}]\,\#, \quad [T\texttt{>}] \to i\,\texttt{<}\,[I\texttt{>}]\,\texttt{>}, \quad [T\,\texttt{::}] \to i\,\texttt{<}\,[I\texttt{>}]\;\texttt{::}, \quad [T] \to i\,\texttt{<}\,[I\texttt{>}].
\end{gathered}$$

The new grammar demonstrates that our initial grammar for C++ identifier expressions is ML(1, 1): it requires contexts of length $k = 1$, and lookahead of length $m = 1$.

[1] We can amend the rules of N to use left-recursion and solve the conflict: $N \to N\,U\;\texttt{::} \mid U\;\texttt{::}$. This correction was made by the Standards Committee in 2003 (see `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#125`). The correction was not motivated by this conflict but by an ambiguity issue, and the fact that the change eliminated the conflict seems to have been a fortunate coincidence. The C++ grammar of the Elsa parser [17] employs right recursion.

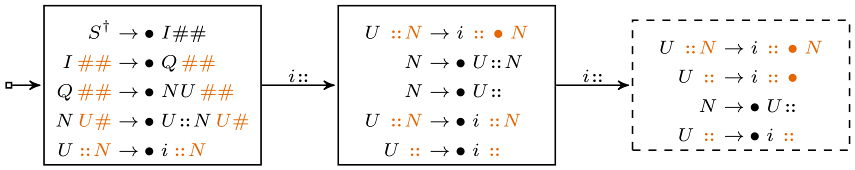
$S^\dagger \to \bullet\ I\#\#$
$I\ \#\# \to \bullet\ Q\ \#\#$
$Q\ \#\# \to \bullet\ NU\ \#\#$
$N\ U\# \to \bullet\ U{::}N\ U\#$
$U\ {::}N \to \bullet\ i\ {::}N$
$i{::}$
$U\ {::}N \to i\ {::}\ \bullet\ N$
$N \to \bullet\ U{::}N$
$N \to \bullet\ U{::}$
$U\ {::}N \to \bullet\ i\ {::}N$
$U\ {::} \to \bullet\ i\ {::}$
$i{::}$
$U\ {::}N \to i\ {::}\ \bullet\ N$
$U\ {::} \to i\ {::}\ \bullet$
$N \to \bullet\ U{::}$
$U\ {::} \to \bullet\ i\ {::}$

Fig. 1. Parts of the uniform ML(2, 0) parser for C++ identifier expressions

Combing Function. Formally, the nonterminals in N' are used in the course of the application of the *uniform k-combing function* comb_k from V^* to $(N' \uplus \Sigma)^*$, defined recursively as:

$$\mathsf{comb}_k(\alpha) = \begin{cases} [A\delta] \cdot \mathsf{comb}_k(\alpha') & \text{if } \alpha = A\delta\alpha', A \in N, \text{ and either } |\delta| = k, \\ & \quad \text{or } |\delta| \le k \text{ and } \alpha' = \varepsilon \\ a \cdot \mathsf{comb}_k(\alpha') & \text{if } \alpha = a\alpha' \text{ and } a \in \Sigma \\ \varepsilon & \text{otherwise, which is if } \alpha = \varepsilon\ . \end{cases}$$

For instance, $\mathsf{comb}_1(ABcDeF) = [AB]c[De][F]$.

The right parts of the rules of $[A\delta]$ are then of the form $\mathsf{comb}_k(\alpha\delta)$ if $A \to \alpha$ was a rule of the original grammar, effectively delaying the reduction of α to A until after δ has been parsed.

Definition 1 (Uniform combing). *Let $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ be a CFG. Its* uniform k-combing *is the CFG $\langle N \cdot V^{\le k}, \Sigma, \{[A\delta] \to \mathsf{comb}_k(\alpha\delta) \mid \delta \in V^{\le k}$ and $A \to \alpha \in P\}, [S]\rangle$.*

Equivalence of the Two Views. Of course we should prove that the two views on ML parsing are equivalent:

Theorem 1. *A grammar is ML(k, m) if and only if the uniform k-combing of its k-extension is LR(m).*

Proof Idea. One can verify that the LR(m) construction on the k-combing of the k-extension of $\mathcal{G}$ and the ML(k, m) construction of Bertsch and Nederhof [4] for the same $\mathcal{G}$ are identical. $\square$

2.3 Selective ML

An issue spotted by Bertsch and Nederhof [4] is that the classes of ML(k, m) grammars and ML(k+1, m) grammars are not comparable: adding further delays can introduce new LR(m) conflicts in the ML($k+1$, m)-transformed grammar.

For instance, the uniform 2-combing of our grammar for C++ identifier expressions is not LR(m) for any m: Fig. 1 shows the path to a conflict similar to that of the original grammar, which is therefore not uniform ML(2, m). Selective ML aims to find the appropriate delay, i.e. the appropriate amount of right context, for each item in the parser, in order to avoid such situations.

Oscillating Behaviour. Bertsch and Nederhof also show that an oscillating behaviour can occur, for instance with the grammar

$$S \to SdA \mid c,\ A \to a \mid ab \qquad (\mathcal{G}_{\text{odd}})$$

being ML(k, 0) only for odd values of k, and the grammar

$$S \to SAd \mid c,\ A \to a \mid ab \qquad (\mathcal{G}_{\text{even}})$$

being ML(k, 0) only for even values of $k > 0$, from which we can build a union grammar

$$S \to SdA \mid SAd \mid c,\ A \to a \mid ab \qquad (\mathcal{G}_2)$$

which is not ML(k, 0) for any k.

Observe however that, if we use different context lengths for the different rules of S in $\mathcal{G}_2$, i.e. if we *select* the different delays, we can still obtain an LR(0) grammar $\mathcal{G}_2'$ with rules

$$\begin{aligned}
[S^\dagger] &\to [S\#]\#,\\
[S\#] &\to [Sd][A\#] \mid [SAd]\# \mid c\#,\\
[Sd] &\to [Sd][Ad] \mid [SAd]d \mid cd,\\
[SAd] &\to [Sd][AAd] \mid [SAd][Ad] \mid c[Ad],\\
[A\#] &\to a\# \mid ab\#,\\
[Ad] &\to ad \mid abd,\\
[AAd] &\to a[Ad] \mid ab[Ad]
\end{aligned} \qquad (\mathcal{G}_2')$$

As we will see, this means that $\mathcal{G}_2$ is selective ML with a delay of at most 2, denoted selML(2, 0). This example shows that selective ML(k, m) is not just about finding a minimal global $k' \leq k$ such that the grammar is uniform ML(k', m). Because the amount of delay is optimized depending on the context, selective ML captures a larger class of grammars.

3 Selective Delays through Grammar Transformation

We define selML(k, m) through a grammar transformation akin to that of Definition 1, but which employs a *combing relation* instead of the uniform k-combing function. We first introduce these relations (Section 3.1) before defining the selML(k, m) grammar class and establishing its relationships with various classes of grammars in Section 3.2 (more comparisons with related work can be found in Section 6).

3.1 Combing Relations

In the following definitions, we let $\mathcal{G} = \langle N, \Sigma, P, S\rangle$ be a context-free grammar. Combing relations are defined through the application of a particular inverse homomorphism throughout the rules of the grammar.

Definition 2 (Selective Combing). *Grammar $\mathcal{G}' = \langle N', \Sigma, P', S' \rangle$ is a* selective combing *of $\mathcal{G}$, denoted $\mathcal{G}$ comb $\mathcal{G}'$, if there exists a homomorphism μ from V'^* to V^* such that*

1. *$\mu(S') = S$,*
2. *$\forall a \in \Sigma, \mu(a) = a$,*
3. *$\mu(N') \subseteq N \cdot V^*$, and*
4. *$\{A' \to \mu(\alpha') \mid A' \to \alpha' \in P'\} = \{A' \to \alpha\delta \mid A' \in N', \mu(A') = A\delta,$ and $A \to \alpha \in P\}$.*

It is a selective k-combing *if furthermore $\mu(N') \subseteq N \cdot V^{\leq k}$.*

We denote the elements of N' by $[A\delta]_i$, such that $\mu([A\delta]_i) = A\delta$, with an i subscript in $\mathbb{N}$ to differentiate nonterminals that share the same image by μ.

Note that, if $\mathcal{G}$ comb $\mathcal{G}'$, then there exists some k such that $\mathcal{G}'$ is a selective k-combing of $\mathcal{G}$, because $\mu(N')$ is a finite subset of $N \cdot V^*$. Another observation is that comb is transitive, and thus we can bypass any intermediate transformation by using the composition of the μ's. In fact, comb is also reflexive (using the identity on N for μ), and is thus a quasi order.

Grammar Cover. It is easy to see that a grammar and all its μ-combings are language equivalent. In fact, we can be more specific, and show that any μ-combing $\mathcal{G}' = \langle N', \Sigma, P', [S]_0 \rangle$ of $\mathcal{G} = \langle N, \Sigma, P, S \rangle$ defines a *right-to-x cover* of $\mathcal{G}$ (see Nijholt [19]), i.e. there exists a homomorphism h from P'^* to P^* such that

1. for all w in $L(\mathcal{G}')$ and right parses π' of w in $\mathcal{G}'$, $h(\pi')$ is a parse of w in $\mathcal{G}$, and
2. for all w in $L(\mathcal{G})$ there is a parse π of w in $\mathcal{G}$, such that there exists a right parse π' of w in $\mathcal{G}'$ with $h(\pi') = \pi$.

Indeed, defining h by

$$h([A\delta]_i \to \alpha) = A \to \mu(\alpha) \cdot \delta^{-1} \tag{1}$$

fits the requirements of a right-to-x cover.

Tree Mapping. Nevertheless, the right-to-x cover characterization is still somewhat unsatisfying, precisely because the exact derivation order x remains unknown. We solve this issue by providing a tree transformation that maps any derivation tree of $\mathcal{G}'$ to a derivation tree of $\mathcal{G}$. Besides allowing us to prove the language equivalence of $\mathcal{G}$ and $\mathcal{G}'$ (see Corollary 1), this transformation also allows us to map any parse tree of $\mathcal{G}'$—the grammar we use for parsing—to its corresponding parse tree of $\mathcal{G}$—the grammar we were interested in in the first place.

We express this transformation as a rewrite system over the set of *unranked forests* $\mathcal{F}(N \cup N' \cup \Sigma)$ over the set of symbols $N \cup N' \cup \Sigma$, defined by the abstract syntax

$$t ::= X(f) \qquad \text{(trees)}$$
$$f ::= \varepsilon \mid f \cdot t \qquad \text{(forests)}$$

where "X" ranges over $N \cup N' \cup \Sigma$ and "$\cdot$" denotes concatenation. Using unranked forests, our tree transformation has a very simple definition, using a rewrite system $\rightarrow_R$ with one rule per nonterminal $[AX_1 \cdots X_r]_i$ in N':

$$[AX_1 \cdots X_r]_i(x_0 \cdot X_1(x_1) \cdots X_r(x_r)) \rightarrow_R A(x_0) \cdot X_1(x_1) \cdots X_r(x_r) \tag{2}$$

with variables $x_0, x_1, \ldots, x_r$ ranging over $\mathcal{F}(N \cup N' \cup \Sigma)$. Clearly, $\rightarrow_R$ is noetherian and confluent, and we can consider the mapping that associates to a derivation tree t in $\mathcal{G}'$ its normal form $t{\downarrow_R}$ (see full paper for details):

Proposition 1. *Let $\mathcal{G}$ be a CFG and $\mathcal{G}'$ a combing of $\mathcal{G}$.*

1. *If t' is a derivation tree of $\mathcal{G}'$, then $t'{\downarrow_R}$ is a derivation tree of $\mathcal{G}$.*
2. *If t is a derivation tree of $\mathcal{G}$, then there exists a derivation tree t' of $\mathcal{G}'$ such that $t = t'{\downarrow_R}$.*

Since $\rightarrow_R$ preserves tree yields, we obtain the language equivalence of $\mathcal{G}$ and $\mathcal{G}'$ as a direct corollary of Proposition 1:

Corollary 1 (Combings Preserve Languages). *Let $\mathcal{G}$ be a k-extended CFG and $\mathcal{G}'$ a combing of $\mathcal{G}$. Then $L(\mathcal{G}) = L(\mathcal{G}')$.*

3.2 Selective ML Grammars

We define selML(k, m) grammars by analogy with the characterization proved in Thm. 1:

Definition 3 (Selective ML). *A grammar is* selML(k, m) *if there exists a selective k-combing of its k-extension that is LR(m).*

Basic Properties. We now investigate the class of selML(k, m) grammars. As a first comparison, we observe that the uniform k-combing of a grammar is by definition a selective k-combing (by setting μ as the identity on $N \cdot V^{\leq k}$), hence the following lemma:

Lemma 1. *If a grammar is ML(k, m) for some k and m, then it is selML(k, m).*

As shown by $\mathcal{G}_2$, this grammar class inclusion is strict.

A second, more interesting comparison holds between selML(0, m) and LR(m). That a LR(m) grammar is selML(0, m) is immediate since comb is reflexive; the converse is not obvious at all, because a 0-combing can involve "duplicated" nonterminals, but holds nevertheless (see full paper for details).

Lemma 2. *A reduced grammar is selML(0, m) if and only if it is LR(m).*

Recall that a context-free language can be generated by some LR(1) grammar if and only if it is deterministic [13], thus selML languages also characterize deterministic languages:

Corollary 2 (Selective ML Languages). *A context-free language has a selML grammar if and only if it is deterministic.*

Proof. Given a selML(k, m) grammar $\mathcal{G}$, we obtain an LR(m) grammar $\mathcal{G}'$ with a deterministic language, and equivalent to $\mathcal{G}$ by Corollary 1. Conversely, given a deterministic language, there exists an LR(1) grammar for it, which is also selML(0,1) by Lemma 2. □

Monotonicity. We should also mention that, unlike uniform ML, increasing k allows strictly more grammars to be captured by selML(k, m). Indeed, if a grammar is a selective k-combing of some grammar $\mathcal{G}$, then it is also a $k+1$-combing using the same μ (with an extra # endmarker), and remains LR(m).

Proposition 2. *If a grammar is selML(k, m) for some k and m, then it is selML(k′, m′) for all $k' \geq k$ and $m' \geq m$.*

Strictness can be witnessed thanks to the grammar family $(\mathcal{G}_3^k)_{k\geq 0}$ defined by

$$S \rightarrow Ac^kA' \mid Bc^kB',\ A \rightarrow cA \mid d,\ B \rightarrow cB \mid d,\ A' \rightarrow cA' \mid a,\ B' \rightarrow cB' \mid b \quad (\mathcal{G}_3^k)$$

where each $\mathcal{G}_3^k$ is selML($k+1$, 0), but not selML(k, m) for any m.

Ambiguity. As a further consequence of Proposition 1, we see that no ambiguous grammar can be selML(k, m) for any k and m.

Proposition 3. *If a grammar is selML(k, m) for some k and m, then it is unambiguous.*

Proof. Assume the opposite: an ambiguous grammar $\mathcal{G}$ has a selective k-combing $\mathcal{G}'$ that is LR(m). Being ambiguous, $\mathcal{G}$ has two different derivation trees t_1 and t_2 with the same yield w. As t_1 and t_2 are in normal form for $\rightarrow_R$, the sets of derivation trees of $\mathcal{G}'$ that rewrite into t_1 and t_2 are disjoint, and using Proposition 1 we can pick two different derivation trees t'_1 and t'_2 with $t_1 = t'_1\downarrow_R$ and $t_2 = t'_2\downarrow_R$. As $\rightarrow_R$ preserves tree yields, both t'_1 and t'_2 share the same yield w, which shows that $\mathcal{G}'$ is also ambiguous, in contradiction with $\mathcal{G}'$ being LR(m) and thus unambiguous. □

Again, this grammar class inclusion is strict, because the following unambiguous grammar for even palindromes is not selML(k, m) for any k or m, since its language is not deterministic:

$$S \rightarrow aSa \mid bSb \mid \varepsilon \qquad (\mathcal{G}_4)$$

Undecidability. Let us first refine the connection between selML and LR in the case of linear grammars: recall that a CFG is *linear* if the right-hand side of each one of its productions contains at most one nonterminal symbol. A consequence is that right contexts in linear CFGs are exclusively composed of terminal symbols. In such a case, the selML(k, m) and LR(k+m) conditions coincide (see full paper for details):

Lemma 3. *Let $\mathcal{G}$ be a reduced linear grammar, and k and m two natural integers. Then $\mathcal{G}$ is selML(k, m) if and only if it is LR($k+m$).*

Note that in the non-linear case, the classes of selML(k, m) and LR(k+m) grammars are incomparable. Nevertheless, we obtain as a consequence of Lemma 3:

Theorem 2. *It is undecidable whether an arbitrary (linear) context-free grammar is selML(k, m) for some k and m, even if we fix either k or m.*

Proof. Knuth [13] has proven that it is undecidable whether an arbitrary linear context-free grammar is LR(n) for some n. □

4 Parser Construction

This section discusses how to directly construct an LR-type parser for a given grammar and fixed k and m values. The algorithm is *incremental*, in that it attempts to use as little right context as possible: this is interesting for efficiency reasons (much as incremental lookaheads in [1, 20]), and actually needed since more context does not necessarily lead to determinism (recall Section 2.3). The class of grammars for which the algorithm terminates successfully (i.e. results in a deterministic parser, without ever reaching a failure state) coincides with the class of selML(k, m) grammars (see Propositions 4 and 5). An extended example of the construction will be given in Section 4.2.

4.1 Algorithm

Algorithm 1 presents the construction of an automaton from the k-extension of a grammar. We will call this the selML(k, m) automaton. In the final stages of the construction, the automaton will resemble an LR(m) automaton for a selective k-combing. Before that, states are initially constructed without right context. Right contexts are extended only where required to solve conflicts.

Items and States. The items manipulated by the algorithm are of form $([A\delta] \rightarrow \alpha \bullet \alpha', L)$, where $L \subseteq \Sigma^{\leq m}$ is a set of terminal lookahead strings, and where α and α' might contain nonterminals of the form $[B\beta]$, where $B \in N$ and $\beta \in V^{\leq k}$. Such nonterminals may later become nonterminals in the selective k-combing of the input grammar. To avoid notational clutter, we assume in what follows that B and $[B]$ are represented in the same way, or equivalently, that an occurrence of B in a right-hand side is implicitly converted to $[B]$ wherever necessary.

States are represented as sets of items. Each such set q is associated with three more sets of items. The first is its closure $\mathsf{close}(q)$. The second is $\mathsf{conflict}(q)$, which is the set of closure items that lead to a shift/reduce or reduce/reduce conflict with another item, either immediately in q or in a state reachable from q by a sequence of transitions. A conflict item signals that the closure step that predicted the corresponding rule, in the form of a non-kernel item, must be reapplied, but now from a nonterminal $[B\beta]$ with longer right context β. Lastly, the set $\mathsf{deprecate}(q)$ contains items that are to be ignored for the purpose of computing the Goto function.

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1]\beta_2, L) \in \mathsf{close}(q)}{([B\beta_1] \to \bullet\, \gamma\beta_1, L') \in \mathsf{close}(q)} \begin{cases} B \to \gamma \in P, \\ L' = \mathsf{First}_m(\beta_2 L) \end{cases} \qquad \text{(closure)}$$

$$\frac{\begin{array}{c}([A_1\delta_1] \to \alpha_1 \bullet \beta_1, L_1) \in \mathsf{close}(q) \\ ([A_2\delta_2] \to \alpha_2 \,\bullet, L_2) \in \mathsf{close}(q)\end{array}}{([A_2\delta_2] \to \alpha_2 \,\bullet, L_2) \in \mathsf{conflict}(q)} \begin{cases} (A_1\delta_1, \alpha_1, \beta_1) \neq (A_2\delta_2, \alpha_2, \varepsilon), \\ \mathsf{First}_m(\mu(\beta_1)L_1) \cap L_2 \neq \emptyset \end{cases} \qquad \text{(conflict detection)}$$

$$\frac{\begin{array}{c}([A\delta] \to \alpha \bullet [B\beta], L) \in \mathsf{close}(q) \\ ([B\beta] \to \bullet\, \gamma, L) \in \mathsf{conflict}(q)\end{array}}{([A\delta] \to \alpha \bullet [B\beta], L) \in \mathsf{conflict}(q)} \qquad \text{(conflict propagation)}$$

$$\frac{\begin{array}{c}([A\delta] \to \alpha \bullet [B\beta_1]X\beta_2, L) \in \mathsf{close}(q) \\ ([B\beta_1] \to \bullet\, \gamma, L') \in \mathsf{conflict}(q)\end{array}}{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)} \begin{cases} |\beta_1| < k, \\ L' = \mathsf{First}_m(X\beta_2 L), \end{cases} \qquad \text{(extension)}$$

$$\frac{([B\beta] \to \bullet\, \gamma, L) \in \mathsf{conflict}(q)}{\bot} \left\{ |\beta| = k \right. \qquad \text{(failure)}$$

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)}{([A\delta] \to \alpha \bullet [B\beta_1]X\beta_2, L) \in \mathsf{deprecate}(q)} \qquad \text{(deprecation)}$$

$$\frac{([A\delta] \to \alpha \bullet [B\beta_1 X]\beta_2, L) \in \mathsf{close}(q)}{([B\beta_1] \to \bullet\, \gamma', L') \in \mathsf{deprecate}(q)} \begin{cases} B \to \gamma \in P, \mu(\gamma') = \gamma\beta_1, \\ L' = \mathsf{First}_m(X\beta_2 L), \end{cases} \qquad \text{(deprecate closure)}$$

Fig. 2. Closure of set q with local resolution of conflicts

Item Closure. The sets $\mathsf{close}(q)$, $\mathsf{conflict}(q)$ and $\mathsf{deprecate}(q)$ are initially computed from the kernel q alone. However, subsequent visits to states reachable from q may lead to new items being added to $\mathsf{conflict}(q)$ and then to $\mathsf{close}(q)$ and $\mathsf{deprecate}(q)$. How items in these three sets are derived from one another for given q is presented as the deduction system in Figure 2.

The *closure* step is performed as in conventional LR parsing, except that right context is copied to the right-hand side of a predicted rule. The *conflict detection* step introduces a conflict item, after a shift/reduce or reduce/reduce conflict appears among the derived items in the closure. Conflict items solicit additional right context, which

– may be available locally in the current state, as in step *extension*, where nonterminal $[B\beta_1]$ is extended to incorporate the following symbol X—we assume μ here is a generic "uncombing" homomorphism, turning a single nonterminal $[B\beta_1]$ into a string $B\beta_1 \in N \cdot V^{\leq k}$—, or
– if no more right context is available at the closure item from which a conflict item was derived, then the closure item itself becomes a conflict item, by step *conflict propagation*—propagation of conflicts across states is realized by Algorithm 1 and will be discussed further below—, or
– if there is ever a need for right context exceeding length k, then the grammar cannot be selML(k, m) and the algorithm terminates reporting failure by step *failure*.

Algorithm 1. Construction of the selML(k, m) automaton for the k-extension of $\mathcal{G} = \langle N, \Sigma, P, S\rangle$, followed by construction of a selective k-combing

```
States $\leftarrow \emptyset$
Transitions $\leftarrow \emptyset$
Agenda $\leftarrow \emptyset$
$q_{\text{init}} = \{(S^{\dagger} \rightarrow \bullet\ S\#^k, \{\varepsilon\})\}$
NewState($q_{\text{init}}$)
while Agenda $\neq \emptyset$ do
    $q \leftarrow$ pop(Agenda)
    remove $(q, X, q')$ from Transitions for any $X$ and $q'$
    apply Figure 2 to add new elements to the three sets associated with $q$
    for all $([A\delta] \rightarrow \alpha X \bullet \beta, L) \in$ conflict($q$) do
        for all $q'$ such that $(q', X, q) \in$ Transitions do
            AddConflict($([A\delta] \rightarrow \alpha \bullet X\mu(\beta), L), q'$)
        end for
    end for
    if there are no $([A\delta] \rightarrow \alpha X \bullet \beta, L) \in$ conflict($q$) then
        $qmax \leftarrow$ close($q$) \ deprecate($q$)
        for all $X$ such that there is $([A\delta] \rightarrow \alpha \bullet X\beta, L) \in qmax$ do
            $q' \leftarrow$ Goto($qmax, X$)
            if $q' \notin$ States then
                NewState($q'$)
            else
                for all $([A'\delta'] \rightarrow \alpha' X \bullet \beta', L) \in$ conflict($q'$) do
                    AddConflict($([A'\delta'] \rightarrow \alpha' \bullet X\mu(\beta'), L), q$)
                end for
            end if
            Transitions $\leftarrow$ Transitions $\cup \{(q, X, q')\}$
        end for
    end if
end while
construct a selective $k$-combing as explained in the running text

function NewState($q$)
    close($q$) $\leftarrow q$
    conflict($q$) $\leftarrow \emptyset$
    deprecate($q$) $\leftarrow \emptyset$
    States $\leftarrow$ States $\cup \{q\}$
    Agenda $\leftarrow$ Agenda $\cup \{q\}$
end function

function AddConflict($([A\delta] \rightarrow \alpha \bullet X\beta, L), q$)
    if $([A\delta] \rightarrow \alpha \bullet X\beta, L) \notin$ conflict($q$) then
        conflict($q$) $\leftarrow$ conflict($q$) $\cup \{([A\delta] \rightarrow \alpha \bullet X\beta, L)\}$
        Agenda $\leftarrow$ Agenda $\cup \{q\}$
    end if
end function
```

Step *deprecation* expresses that an item with shorter right context is to be ignored for the purpose of computing the Goto function. The Goto function will be discussed further below. Similarly, step *deprecate closure* expresses that all items predicted from the item with shorter right context are to be ignored.

Main Algorithm. Initially, the agenda contains only the initial state, which is added in line 5. Line 7 of the algorithm removes an arbitrary element from the agenda and assigns it to variable q. At that point, either $\mathsf{close}(q) = q$ and $\mathsf{conflict}(q) = \mathsf{deprecate}(q) = \emptyset$ if q was not considered by line 7 before, or elements may have been added to $\mathsf{conflict}(q)$ since the last such consideration, which also requires updating of $\mathsf{close}(q)$ and $\mathsf{deprecate}(q)$, by Figure 2. By a change of the latter two sets, also the outgoing transitions may change. To keep the presentation simple, we assume that all outgoing transitions are first removed (on line 8) and then recomputed. From line 10, conflicting items are propagated to states immediately preceding the current state, by one transition. Such a preceding state is then put on the agenda so that it will be revisited later.

Outgoing transitions are (re-)computed from line 15 onward. This is only done if no conflicting items had to be propagated to preceding states. Such conflict items would imply that q itself will not be reachable from the initial state in the final automaton, and in that case there would be no benefit in constructing outgoing transitions from q.

For the purpose of applying the Goto function, we are only interested in the closure items that have maximal right context, as all items with shorter context were found to lead to conflicts. This is the reason why we take the set difference $qmax = \mathsf{close}(q) \setminus \mathsf{deprecate}(q)$. The Goto function is defined much as usual:

$$\mathsf{Goto}(qmax, X) = \{([A\delta] \to \alpha X \bullet \beta, L) \mid ([A\delta] \to \alpha \bullet X\beta, L) \in qmax\} \ . \qquad (3)$$

The loop from line 22 is very similar to that from line 10. In both cases, conflicting items are propagated from a state q_2 to a state q_1 along a transition (q_1, X, q_2). The difference lies in whether q_1 or q_2 is the currently popped element q in line 7. The propagation must be allowed to happen in both ways, as it cannot be guaranteed that no new transitions are found leading to states at which conflicts have previously been processed.

Combing Construction. After the agenda in Algorithm 1 becomes empty, only those states reachable from the initial state q_{init} via transitions in Transitions are relevant, and the remaining ones can be removed from States. From the reachable states, we can then construct a selective k-combing, with start symbol $S^\dagger$, as follows.

For each $q_n \in$ States and $([A\delta] \to X_1 \cdots X_n \bullet, L) \in \mathsf{close}(q_n) \setminus \mathsf{deprecate}(q_n)$, some $n \geq 0$, find each choice of:

- $q_0, \ldots, q_{n-1}$,
- $\beta_0, \ldots, \beta_n$, with $\beta_n = \varepsilon$,

such that for $0 \leq j < n$,

- $(q_j, X_{j+1}, q_{j+1}) \in \mathsf{Transitions}$,
- $([A\delta] \to X_1 \cdots X_j \bullet X_{j+1}\beta_{j+1}, L) \in \mathsf{close}(q_j) \setminus \mathsf{deprecate}(q_j)$, and
- $\beta_j = \mu(X_{j+1})\beta_{j+1}$.

It can be easily seen that β_0 must be of the form $\alpha\delta$, for some rule $A \to \alpha$. For each choice of the above, now create a rule $Y_0 \to Y_1 \cdots Y_n$, where Y_0 stands for the triple $(q_0, A\delta, L)$, and for $1 \leq j \leq n$:

- if X_j is a terminal then $Y_j = X_j$, and
- if X_j is of the form $[B_j\gamma_j]$ then Y_j stands for the triple $(q_{j-1}, B_j\gamma_j, L_j)$, where $L_j = \mathsf{First}_m(\beta_j L)$.

We assume here that $\mu(Y_0) = A\delta$ and $\mu(Y_j) = B_j\gamma_j$ for $1 \leq j \leq n$.

4.2 Example

Example 2. Let us apply Algorithm 1 to the construction of a selML(2, 0) parser for $\mathcal{G}_{\text{odd}}$. The initial state is $q_{\text{init}} = \{S^\dagger \to \bullet S\#\#\}$ (there is no lookahead set since we set $m = 0$) and produces through the rules of Fig. 2

$$\mathsf{close}(q_{\text{init}}) = \{S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c\} . \tag{4}$$

Fast-forwarding a little, the construction eventually reaches state $q_{Sd} = \{S \to Sd \bullet A\}$ with

$$\mathsf{close}(q_{Sd}) = \{S \to Sd \bullet A, A \to \bullet a, A \to \bullet ab\} , \tag{5}$$

which in turn reaches state $q_{Sda} = \{A \to a\bullet, A \to a \bullet b\}$ with

$$\mathsf{close}(q_{Sda}) = q_{Sda} , \tag{6}$$
$$\mathsf{conflict}(q_{Sda}) = \{A \to a\bullet\} . \tag{7}$$

As this item is marked as a conflict item, line 10 of Algorithm 1 sets

$$\mathsf{conflict}(q_{Sd}) = \{A \to \bullet a\} , \tag{8}$$

and puts q_{Sd} back in the agenda. Then, the *conflict propagation* rule is fired to set

$$\mathsf{conflict}(q_{Sd}) = \{A \to \bullet a, S \to Sd \bullet A\} , \tag{9}$$

and by successive backward propagation steps we get

$$\mathsf{conflict}(q_{\text{init}}) = \{S \to \bullet SdA\} . \tag{10}$$

The *extension* rule then yields

$$\mathsf{close}(q_{\mathrm{init}}) = \{S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A\}\ , \tag{11}$$

which is closed to obtain

$$\begin{aligned}\mathsf{close}(q_{\mathrm{init}}) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A,\\ &[S\#] \to \bullet SdA\#, [S\#] \to \bullet c\#, [Sd] \to \bullet SdAd, [Sd] \to \bullet cd\}\ ,\end{aligned} \tag{12}$$

and we can apply again the *extension* rule with the conflicting item $S \to \bullet SdA$:

$$\begin{aligned}\mathsf{close}(q_{\mathrm{init}}) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, S^\dagger \to \bullet[S\#]\#, S \to \bullet[Sd]A,\\ &[S\#] \to \bullet SdA\#, [S\#] \to \bullet c\#, [Sd] \to \bullet SdAd, [Sd] \to \bullet cd,\\ &[S\#] \to \bullet[Sd]A\#, [Sd] \to \bullet[Sd]Ad\}\ .\end{aligned} \tag{13}$$

The *deprecate* and *deprecate closure* rules then yield

$$\begin{aligned}\mathsf{deprecate}(q) = \{&S^\dagger \to \bullet S\#\#, S \to \bullet SdA, S \to \bullet c, [S\#] \to \bullet SdA\#,\\ &[Sd] \to \bullet SdAd, \ldots\}\ .\end{aligned} \tag{14}$$

We leave the following steps to the reader; the resulting parser is displayed in Fig. 3 (showing only items in $\mathsf{close}(q) \setminus \mathsf{deprecate}(q)$ in states).

4.3 Correctness

First observe that Algorithm 1 always terminates: the number of possible sets q, along with the growing sets $\mathsf{close}(q)$, $\mathsf{conflict}(q)$ and $\mathsf{deprecate}(q)$, is bounded.

Termination by the *failure* step of Fig. 2 occurs only when we know that the resulting parser cannot be deterministic; conversely, successful termination means that a deterministic parser has been constructed. One could easily modify the construction to keep running in case of failure and output a nondeterministic parser instead, for instance to use a generalized LR parsing algorithm on the obtained parser.

The correctness of the construction follows from Propositions 4 and 5 (see full paper for details).

Proposition 4. *If Algorithm 1 terminates successfully, then the constructed grammar is a selective k-combing. Furthermore, this combing is LR(m).*

Proof Idea. The structure of the selML(k, m) automaton and the item sets ensure that the constructed grammar satisfies all the requirements of a selective k-combing. Had this been non-LR(m), then there would have been further steps or failure. □

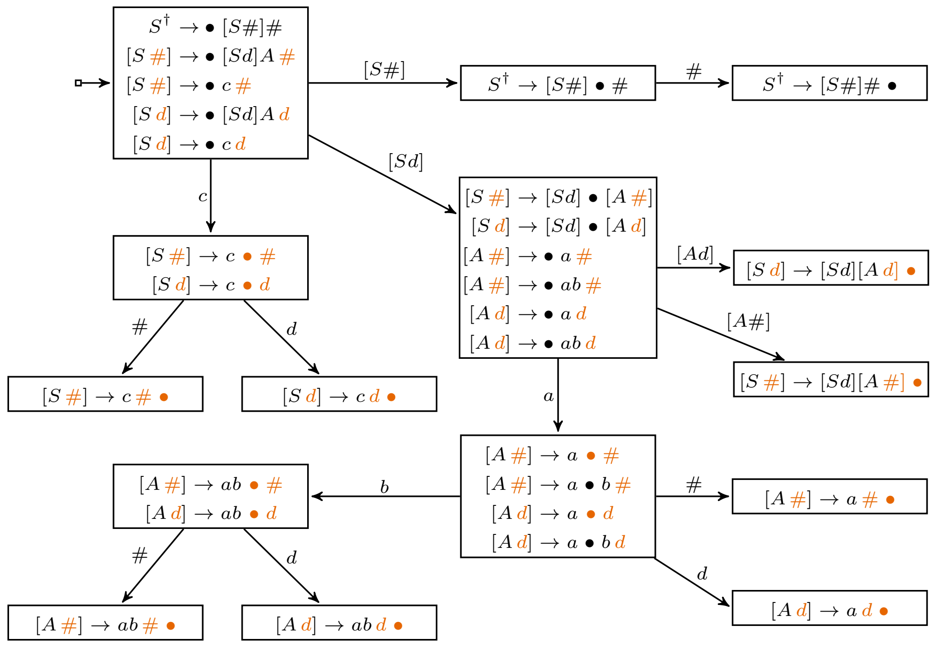
$S^\dagger \to \bullet\ [S\#]\#$
$[S\ \#] \to \bullet\ [Sd]A\ \#$
$[S\ \#] \to \bullet\ c\ \#$
$[S\ d] \to \bullet\ [Sd]A\ d$
$[S\ d] \to \bullet\ c\ d$
$[S\#]$
$S^\dagger \to [S\#] \bullet \#$
$\#$
$S^\dagger \to [S\#]\# \bullet$
$[Sd]$
c
$[S\ \#] \to c \bullet \#$
$[S\ d] \to c \bullet d$
$[S\ \#] \to [Sd] \bullet [A\ \#]$
$[S\ d] \to [Sd] \bullet [A\ d]$
$[A\ \#] \to \bullet\ a\ \#$
$[A\ \#] \to \bullet\ ab\ \#$
$[A\ d] \to \bullet\ a\ d$
$[A\ d] \to \bullet\ ab\ d$
$[Ad]$
$[S\ d] \to [Sd][A\ d] \bullet$
$[A\#]$
$[S\ \#] \to [Sd][A\ \#] \bullet$
$\#$
d
$[S\ \#] \to c\ \# \bullet$
$[S\ d] \to c\ d \bullet$
a
$[A\ \#] \to a \bullet \#$
$[A\ \#] \to a \bullet b\ \#$
$[A\ d] \to a \bullet d$
$[A\ d] \to a \bullet b\ d$
b
$[A\ \#] \to ab \bullet \#$
$[A\ d] \to ab \bullet d$
$\#$
$[A\ \#] \to a\ \# \bullet$
d
$[A\ d] \to a\ d \bullet$
$\#$
d
$[A\ \#] \to ab\ \# \bullet$
$[A\ d] \to ab\ d \bullet$

Fig. 3. The selML(2, 0) parser for $\mathcal{G}_{\text{odd}}$

Proposition 5. *If the grammar is selML(k, m), then the algorithm terminates successfully.*

Proof Idea. The selML(k, m) automaton under construction reflects minimum right context for nonterminal occurrences in any selective k-combing with the LR(m) property. Furthermore, failure would imply that right context of length exceeding k is needed. □

As a consequence, we can refine the statement of Theorem 2 with

Corollary 3. *It is decidable whether an arbitrary context-free grammar is selML(k, m), for given k and m.*

5 Experimental Results

We have implemented a proof of concept of Algorithm 1, which can be downloaded from `http://www.cs.st-andrews.ac.uk/~mjn/code/mlparsing/`. Its purpose is not to build actual parsers for programming languages, but merely to check the feasibility of the approach and compare selML with uniform ML and more classical parsers.

Table 2. Results on example grammars

	$\lvert N\rvert$	$\lvert P\rvert$	LR classes: #states	ML classes: #states
Example 3	2	6	non-LR(m)	ML(3,1): 357, selML(3,1): 41, ML(2,2): 351, selML(2,2): 77
Example 4	5	8	LR(2): 16	(sel)ML(1,0): 17
Example 5	3	5	LALR(1): 11	ML(1,0): 17, selML(1,0): 15

Grammar Collection. We investigated a set of *small* grammars that exhibit well-identified syntactic difficulties, to see whether they are treated correctly by a given parsing technique, or lie beyond its grasp. This set of grammars was compiled by Basten [2] and extended in [22], containing mostly grammars for programming languages from the parsing literature and the comp.compilers archive, but also a few RNA grammars used by the bioinformatics community [21].

Conflicts. As expected, we identified a few grammars that were not LALR(1) but were selML(k, m) for small values of k and m. Results are summarized in Table 2.

Example 3 (Tiger). One such example is an excerpt from the Tiger syntax found at `http://compilers.iecc.com/comparch/article/98-05-030`. The grammar describes assignment expressions E, which are typically of the form "$L := E$" for L an lvalue.

$$E \to L \mid L := E \mid i[E] \text{ of } E \qquad\qquad L \to i \mid L[E] \mid L.i$$

The grammar is not LR(m) for any m, but is ML(3, 1) and ML(2, 2): a conflict arises between inputs of the form "$i[E]$ of E" and "$i[E] := E$", where the initial i should be kept as such and the parser should shift in the first case, and reduce to L in the second case. An ML(3, 1) or ML(2, 2) parser scans up to the "of" or ":=" token that resolves this conflict, across the infinite language generated by E.

Example 4 (Typed Pascal Declarations). Another example is a version of Pascal identifier declarations with type checking performed at the syntax level, which was proposed by Tai [26]. The grammar is LR(2) and ML(1,0):

$$\begin{array}{ll} D \to \text{var}\, IL\, IT\, ; \mid \text{var}\, RL\, RT\, ; & \\ IL \to i\, ,\, IL \mid i & IT \to :\ \text{integer} \\ RL \to i\, ,\, RL \mid i & RT \to :\ \text{real} \end{array}$$

On an input like "var i , i , i : real ;", a conflict arises between the reductions of the last identifier i to either an integer list IL or a real list RL with ":" as terminal lookahead. By delaying these reductions, one can identify either an integer type IT or a real type RT.

Non-Monotonicity. We found that non-monotonic behaviour with uniform ML parsers occurs more often than expected. Here is one example in addition to the C++ example given in Section 2.1; more could be found in particular with the RNA grammars of Reeder et al. [21].

Example 5 (Pascal Compound Statements). The following is an excerpt from ISO Pascal and defines compound statements C in terms of ";"-separated lists of statements S:

$$C \to \text{begin } L \text{ end} \qquad\qquad L \to L \ ; \ S \mid S \qquad\qquad S \to \varepsilon \mid C$$

This is an LALR(1) and ML(1, 0) grammar, but it is not ML(2, 0): the nonterminal $[L;S]$ has a rule $[L;S] \to [L;S];[S]$, giving rise to a nonterminal $[S]$ with rules $[S] \to \varepsilon \mid [C]$ and a shift/reduce conflict—in fact, this argument shows more generally that the grammar is not ML(k, 0) for even k.

Parser Size. Because selML parsers introduce new context symbols only when required, they can be smaller than the corresponding LR or uniform ML parsers, which carry full lookahead lengths in their items—this issue has been investigated for instance by Ancona et al. [1] and Parr and Quong [20] for LR and LL parsers. Our results are inconclusive as to the difference of parser size (in terms of numbers of states) between selML and LR. However, selML parsers tend to be considerably smaller than uniform ML parsers. Compare, for example, the numbers of states in the case of ML and selML for Example 3, in Table 2.

In fact, we can make the argument more formal: consider the family of grammars $(G_4^j)_{j>0}$, each with rules:

$$\begin{aligned} &S \to A \mid D, \quad A \to a \mid Ab \mid Ac, \quad D \to EF^{j-1}F \mid E'F^{j-1}F', \\ &F \to a \mid bF, \quad F' \to f \mid bF', \qquad E \to e, \quad E' \to e\ . \end{aligned} \tag{G_4^j}$$

The uniform ML(j, 0) parser for G_4^j has exponentially many states in j, caused by the rules $[Aw] \to aw$ for all w in $\{b,c\}^j$, while the selective ML(j, 0) parser has only a linear number of states, as there is no need for delays in that part of the grammar.

6 Related Work

Grammar Transformations and Coverings. The idea of using grammar transformations to obtain LR(1) or even simpler grammars has been thoroughly investigated in the framework of *grammar covers* [19]. Among the most notable applications, Mickunas et al. [18] provide transformations from LR(k) grammars into much simpler classes such as *simple LR(1)* or *(1,1)-bounded right context*; Soisalon-Soininen and Ukkonen [24] transform *predictive LR(k)* grammars into LL(k) ones by generalizing the notion of left-corner parsing. Such techniques were often limited however to *right-to-right* or *left-to-right* covers, whereas our transformation is not confined to such a strict framework.

Parsing with Delays. A different notion of delayed reductions was already suggested by Knuth [13] and later formalized by Szymanski and Williams [25] as *LR(k, t) parsing*, where one of the t leftmost phrases in any rightmost derivation can be reduced using a lookahead of k symbols. The difference between the two notions of delay can be witnessed with linear grammars, which are LR(k, t) if and only if they are LR(k)—because there is always at most one phrase in a derivation—but selML(k, m) if and only if they are LR($k + m$)—as shown in Lemma 3.

Like selML languages, and unlike more powerful noncanonical classes, the class of LR(k, t) grammars characterizes deterministic context-free languages. The associated parsing algorithm is quite different however from that of selML parsing: it uses the two-stacks model of noncanonical parsing, where reduced nonterminals are pushed back at the beginning of the input to serve as lookahead in reductions deeper in the stack. Comparatively, selML parsing uses the conventional LR parsing tables with a single stack.

Selectivity. Several parser construction methods attempt to use as little "information" as possible before committing to a parsing action: Ancona et al. [1] and Parr and Quong [20] try to use as little lookahead as possible in LR(k) or LL(k) parsing, Demers [7] generalizes left-corner parsing to delay decisions possibly as late as an LR parser, and Fortes Gálvez et al. [9] propose a noncanonical parsing algorithm that explores as little right context as possible.

7 Concluding Remarks

Selective ML parsing offers an original balance between

– enlarging the class of admissible grammars, compared to LR parsing, while
– remaining a deterministic parsing technique, with linear-time parsing and exclusion of ambiguities,
– having a simple description as a grammar transformation, and
– allowing the concrete construction of LR parse tables.

This last point is also of interest to practitioners who have embraced general, nondeterministic parsing techniques [12]: unlike noncanonical or regular-lookahead extensions, selML parsers can be used for nondeterministic parsing exactly like LR parsers. Having fewer conflicts than conventional LR parsers, they will resort less often to nondeterminism, and might be more efficient.

References

1. Ancona, M., Dodero, G., Gianuzzi, V., Morgavi, M.: Efficient construction of LR(k) states and tables. ACM Trans. Progr. Lang. Syst. 13(1), 150–178 (1991)
2. Basten, H.J.S.: The usability of ambiguity detection methods for context-free grammars. In: Electronic Notes in Theoretical Computer Science, LDTA 2008, vol. 238, pp. 35–46. Elsevier (2008)
3. Bermudez, M.E., Schimpf, K.M.: Practical arbitrary lookahead LR parsing. J. Comput. Syst. Sci. 41(2), 230–250 (1990)

4. Bertsch, E., Nederhof, M.J.: Some observations on LR-like parsing with delayed reduction. Inf. Process. Lett. 104(6), 195–199 (2007)
5. Boullier, P.: Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques. Thèse d'État, Université d'Orléans (1984)
6. Čulik, K., Cohen, R.: LR-Regular grammars—an extension of LR(k) grammars. J. Comput. Syst. Sci. 7(1), 66–96 (1973)
7. Demers, A.J.: Generalized left corner parsing. In: POPL 1977, pp. 170–182. ACM (1977)
8. Farré, J., Fortes Gálvez, J.: A Bounded Graph-Connect Construction for LR-regular Parsers. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 244–258. Springer, Heidelberg (2001)
9. Gálvez, J.F., Schmitz, S., Farré, J.: Shift-Resolve Parsing: Simple, Unbounded Lookahead, Linear Time. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 253–264. Springer, Heidelberg (2006)
10. Gosling, J., Joy, B., Steele, G.: The Java$^{\text{TM}}$ Language Specification, 1st edn. Addison-Wesley (1996)
11. ISO: ISO/IEC 14882:1998: Programming Languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
12. Kats, L.C., Visser, E., Wachsmuth, G.: Pure and declarative syntax definition: Paradise lost and regained. In: OOPSLA 2010, pp. 918–932. ACM (2010)
13. Knuth, D.E.: On the translation of languages from left to right. Inform. and Cont. 8(6), 607–639 (1965)
14. Leermakers, R.: Recursive ascent parsing: from Earley to Marcus. Theor. Comput. Sci. 104(2), 299–312 (1992)
15. Malloy, B.A., Power, J.F., Waldron, J.T.: Applying software engineering techniques to parser design: the development of a C# parser. In: SAICSIT 2002, pp. 75–82. SAICSIT (2002)
16. Marcus, M.P.: A Theory of Syntactic Recognition for Natural Language. Series in Artificial Intelligence. MIT Press (1980)
17. McPeak, S., Necula, G.C.: Elkhound: A Fast, Practical GLR Parser Generator. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 73–88. Springer, Heidelberg (2004)
18. Mickunas, M.D., Lancaster, R.L., Schneider, V.B.: Transforming LR(k) grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context grammars. J. ACM 23(3), 511–533 (1976)
19. Nijholt, A.: Context-free grammars: Covers, normal forms, and parsing. LNCS, vol. 93. Springer (1980)
20. Parr, T.J., Quong, R.W.: LL and LR translators need $k > 1$ lookahead. ACM Sigplan. Not. 31(2), 27–34 (1996)
21. Reeder, J., Steffen, P., Giegerich, R.: Effective ambiguity checking in biosequence analysis. BMC Bioinformatics 6, 153 (2005)
22. Schmitz, S.: An experimental ambiguity detection tool. Science of Computer Programming 75(1-2), 71–84 (2010)
23. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, vol. II: LR(k) and LL(k) Parsing. EATCS Monographs on Theoretical Computer Science, vol. 20. Springer (1990)
24. Soisalon-Soininen, E., Ukkonen, E.: A method for transforming grammars into LL(k) form. Acta Inf. 12(4), 339–369 (1979)
25. Szymanski, T.G., Williams, J.H.: Noncanonical extensions of bottom-up parsing techniques. SIAM J. Comput. 5(2), 231–250 (1976)
26. Tai, K.C.: Noncanonical SLR(1) grammars. ACM Trans. Progr. Lang. Syst. 1(2), 295–320 (1979)