mathematics

Article

On Parsing Programming Languages with Turing-Complete Parser

Boštjan Slivnik [1,†] and Marjan Mernik [2,*,†]

[1] Faculty of Computer and Information Science, University of Ljubljana, Večna Pot 113, 1000 Ljubljana, Slovenia
[2] Faculty of Electrical Engineering and Computer Science, University of Maribor, Koroška Cesta 46, 2000 Maribor, Slovenia
* Correspondence: marjan.mernik@um.si
† These authors contributed equally to this work.

Abstract: A new parsing method based on the semi-Thue system is described. Similar to, but with more efficient implementation than Markov normal algorithms, it can be used for parsing any recursively enumerable language. Despite its computational power, it is meant to be used primarily for parsing programming and domain-specific languages. It enables a straightforward simulation of a number of existing parsing algorithms based on context-free grammars. The list includes both top-down shift-produce methods (such as SLL and LL) and bottom-up shift-reduce methods (such as LALR and LR), as well as mixed top-down-and-bottom-up methods such as LLLR. To justify the use of the new parsing method, the paper provides numerous examples of how a parser can actually be made in practice. It is advised that the main part of the parser is based on some simple well-established approach, e.g., SLL(1), while syntactically more complicated phrases can be parsed by exploiting the full power of the new parser. These phrases may either be extensions to the original language or some embedded domain-specific language. In all such and similar cases, no part of the language is restricted to be context-free. In fact, context-sensitive languages can be handled quite efficiently.

Keywords: turing-complete parsing; context-sensitive; error recovery

MSC: 68Q45

Citation: Slivnik, B.; Mernik, M. On Parsing Programming Languages with Turing-Complete Parser. *Mathematics* **2023**, *11*, 1594. https://doi.org/10.3390/math11071594

Academic Editor: Francesco Calimeri

Received: 31 January 2023
Revised: 18 March 2023
Accepted: 23 March 2023
Published: 25 March 2023

1. Introduction

Many of the most widely used compilers nowadays typically use hand-coded recursive-descent parsers. Good examples are GCC C/C++ compilers, which initially used `bison`-generated LALR parsers, and Clang/LLVM C/C++ compilers. The two most often cited reasons are as follows. First, as recursive-descent parsers perform top-down parsing, it is known not only which phrase is being parsed at each moment, but also its position in the abstract syntax tree. Thus, it is usually easier to generate precise error messages and implement good error recovery techniques in recursive-descent parsers than in bottom-up parsers. Second, a programming language usually contains only a few constructs that are hard to parse. For instance, when a programming language is extended, new constructs are often syntactically more complicated than the existing ones, as these must not be altered to maintain backward compatibility—Java was initially specified by an LALR grammar [1], but after the first extensions were introduced, the grammar in the specification ceased to adhere to LALR requirements [2]. However, various modifications or even hacks, e.g., the local lookahead extension, can be much easier added to a hand-coded recursive-descent parser than to a generated one [3].

Not even ANTLR, perhaps the most advanced parser generator, can fulfill the second of the two requirements fully. Based on ALL(*) parsing [4], it cannot handle, for instance, mutually left recursive symbols or indirect left recursion [5]. Hence, a parser, which provides a good error recovery and is capable of parsing even the most complicated constructs, is needed. To address all these issues, the paper provides the following contributions:

1. *Formulation of a new parsing model called the* LLR*-system.* It is a Turing-complete rewriting (semi-Thue) system similar to Markov normal algorithms [6,7] on the one hand and to a context-sensitive reduction system [8] on the other hand. Its computational power enables parsing far beyond the limit of context-free languages, but, in the context of parsing programming and domain-specific languages [9,10], it can (i) deal with the most complex language constructs and (ii) to enable the implementation of parser functionalities that would otherwise be implemented by augmenting or modifying the parser source code, i.e., outside the formalism on which the parser is based.
2. *Representation of existing parsing algorithms in the* LLR*-system*. It is demonstrated how canonical LL and LR parsing and their simplified variants, namely SLL and LALR parsing, can be implemented using the LLR-system. Furthermore, it is demonstrated how bidirectional parsing, i.e., a mixture of top-down and bottom-up parsing similar to LLLR parsing [11,12], can be implemented in the most natural way. This makes many important patterns needed to build language recognizers readily available.
3. *Modelling and implementation of error recovery*. It is shown how two of the most widely used techniques, namely the panic-mode and phrase-level error recovery, can be described naturally as a part of the LLR-system. Furthermore, as the LLR-system can start parsing in the middle of a sentential form (as demonstrated in WEB and CWEB [13,14]), it can simply restart parsing past the point of a syntax error if other error recovery methods fail.

The LLR-system, defined here for the first time, is basically a very simple method, but, just as any other parsing method, it requires some expertise to implement a parser. This paper thus focuses (a) on the definition of the LLR-system; (b) on providing examples that help one to start gaining skills in implementing parsers based on the LLR-system, as its underlying principle and rules are different from those of context-free grammars; and (c) on demonstrating that, once implemented, the LLR-system is nevertheless an efficient language recognizer, despite being a more general and powerful computation model than context-free grammars and pushdown automata.

This paper assumes that the reader shares a good knowledge of context-free parsing as exposed by, for instance, Sippu and Soisalon-Soininen [15,16] or Grune and Jacobs [17]. After the section on related work, the core of the paper starts with Section 3, containing the definition of the LLR-system. In Section 4, a formulation of the aforementioned parsing algorithms in the LLR-system is described and supported by examples. Implementation of error recovery is described in Section 5. In Section 6, the efficiency of parsers based on LLR-systems is demonstrated on a real programming language and source files. After the conclusion, the appendix contains a proof, and then the LLR-system is Turing-complete.

2. Related Work

As mentioned above, the two models most similar to the LLR-system in terms of how the parser is specified are Markov normal algorithms [6,7] and the context-sensitive reduction system [8]. Because of their similarity, they will be compared to the LLR-system in Section 3, where the LLR-system is defined. However, as the LLR-system is Turing-complete, it is, to some extent, similar to hand-coded parsers, as they can most easily be augmented with an auxiliary code implementing parts of a parser that require computational strength beyond any context-free language formalism. Among hand-coded parsers, the recursive-descent implementation of the SLL(1) parsers for context-free languages is the most widely used example of this kind [15,17,18].

In general, the syntax of programming and most domain-specific languages is described by context-free grammars [19], while common non-context-free features of these languages, e.g., scopes and namespaces, are usually dealt with throughout a lexical, syntax and semantic analysis [18,20,21]. These features are dealt with informal techniques as standard lexers, or the context-free parser can be augmented with context-aware lexers [22–24].

Another way is using context-free parsers extended with a subset of approaches used for parsing specific types of non-context-free language constructs and features. [25]. One such approach is stateful parsing, which maintains a global parse-wide state. Whenever a context-free formalism proves to be too weak, a stateful parser compensates its weakness by manipulating the state. This can be conducted within a hand-written parser, within a `yacc`-generated parser [26], or if parser combinators are used. [27,28].

ANTLR4 [5], which is one of the most popular and advanced parser generators nowadays, implements ALL$(*)$, i.e., adaptive LL parsing, and provides support for scannerless parsers and predicated ALL$(*)$ grammars [4]. Its scannerless parsers are useful for resolving context-sensitive lexical issues. Predicated ALL$(*)$ grammars use (a) side-effect-free semantic predicates to ensure a particular production can be applied in a given context and (b) mutators, to alter the sentential form, if needed. Strictly speaking, a predicated ALL$(*)$ grammar generates "a recursively enumerable language because each mutator can be a Turing machine. In reality, grammar writers do not use this generality so it is standard practice to consider the language class to be the context-sensitive languages instead" [4]. ANTLR4 can express a non-context-free language using semantic predicates and mutators. These must be implemented in a host language and are therefore an extension of the formalism, not a part of it.

Stateful parsers can be used for parsing non-context-free languages, but a lot of discipline is required to write and maintain them. One example of making a stateful parser manageable in practice is adding backtracking to LR parsing, in order to make a parsing of ambiguous context-dependent languages, e.g., C++, possible [29]. It takes a lot of effort to parse context-dependent language features using GLR and GLL parsing, because multiple copies of global data structures must be maintained. To avoid this, three kinds of semantic actions are used in the backtracking of LR parsing: *trial* actions for directing future parsing, *undo* actions for reverting side-effects of trial actions, and *final* actions for reductions that can never be undone [29]. As above, context-sensitive constructs are not formulated with grammar productions but with semantic actions expressed in a host language. At each step, such parsers can select the next action by only using the part of the input that has been parsed so far, while the proper context-sensitive parsing is not bound by this limitation [8].

In principled stateful parsing [30], the idea of using a number of predefined primitive operations for inspecting and altering a mutable parse-wide state is used. This operation focuses mostly on dealing with a set of specific context-sensitive features, including the namespace classification, handling of whitespace, and alike, but they may prove useful in general. Similar to ANTLR's predicated grammars or principled stateful parsing, data-dependent grammars [31,32] use semantic values to augment productions. If the GLL parsing algorithm is used with data-dependent grammars, context-sensitive lexical problems and indentation-sensitive rules can be dealt with effectivelly [33].

Some approaches are stronger than classical algorithms for parsing context-free languages. One such approach is based on the parsing expression grammars (PEGs): productions in PEGs are ordered similarly to Markov normal algorithms. PEG parsing can parse some non-context-free languages and it is also possible "to construct a PEG language which is complete for P under log-space reductions", but even-palindromes cannot be parsed using a PEG parser (meanwhile, obviously, they can be generated by a context-free grammar) [34]. A memoizing PEG parser called Rats! [35] expresses rules witnin transactions. Hence, the state changes (as in stateful parsing) can be undone whenever certain conditions cannot be satisfied; meanwhile, Nez, another PEG parser generator, focuses on the declarative parser specifications [36–38]. Likewise, parser combinators "are a middle ground between the fine control of hand-rolled parsers and the high-level almost grammar-like appearance of parsers created via parser generators" [28]. Moreover, monadic parser combinators can deal with context-sensitive language constructs [39]. The global state, however, needs to be sent around the parser code during parsing. Hence, adding the reduction automaton to the monadic parser would most likely result in a significantly more complex implementation of the latter.

In contrast with the computational power of the LLR-system (or the context-sensitive reduction system), most authors focus on typical concrete non-context-free features found in most programming languages. The list of such features includes the indentation sensitivity of Python or Haskell, typedef names in C/C++, HERE document in Perl, user-defined operators with custom precedence, and associativity or Ruby [30,31,33,36,37]. However, the true general context-sensitive parsing algorithms are rare: one is a general context-sensitive parsing algorithm that finds each derivation only once [40] and the other is a CYK-like tabular algorithm for context-sensitive languages [41]. To reduce the complexity of context-sensitive parsing, it was suggested that weakly context-sensitive languages [42] and loop-free context-sensitive languages [43] could be used as models for parsing programming languages.

The summary of the most important approaches for parsing non-context-free languages is shown in Table 1. Note, however, than many approaches could be included into the right column if semantic rules are misused for parsing. For instance, as non-context-free constructs can be parsed by a user-supplied code within a semantic action in, say, `yacc` or `bison`, these two tools could, in principle, be included into the topmost boxes in the right column of Table 1.

Table 1. Summary of different approaches for parsing non-context-free languages.

Strength	**Supported by the Formalism**	**If Augmented by Code**
Turing-complete	Markov normal algorithms [6,7] LLR-systems	Predicated ALL($*$) [4] (if mutators are abused)
Context-sensitive	Woods algorithm [40] CYK-like tabular algorithm [41] CS reduction systems [8]	
Limited context-sensitive	Weakly context-sensitive languages [42] Loop-free context-sensitive languages [43] ALL($*$) [4,5] PEG parsing [34] Context-aware lexers [22–24]	Backtracking LR [29] (can be done safely) Principled stateful parser [30] Data-dependent grammars [31,32]

However, most parser algorithms used nowadays are descendants of relatively old algorithms, e.g., the canonical LR and LL parsing [44,45] or even PEG parsing [46]. However, although extensively studied [17,18], error recovery has never been formalized to the same extend as parsing. After all, actual errors made by programmers are hard to formalize and any definition would most likely not match the definition of a syntax error as detected by a particular parser [16]. Even more, most error recovery methods are either automatic or implemented outside the formalism used for specifying the syntax of a language being parsed, or both.

In practice, the error recovery approach for predictive top-down parsers made popular by Wirth [47] resulted in two most widely used methods: panic-mode error recovery and phrase-level error-recovery. Two well-known LALR parser generators, namely `yacc` and `bison`, provide an error token to support simple and rather inflexible panic-mode error recovery [26]. The ANTLR4 parser generator performs automatic error recovery, which supports a single symbol insertion or deletion [4], i.e., a combination of both methods, but occasionally requires that the entire input is reparsed and offers the parser writer little authority over error recovery actions.

Various parsing libraries often lack the support for error recovery. One library that does care about it, namely Parglare, implements panic-mode error recovery for its LR and GRL parsers [48]. Furthermore, it allows replacing panic-mode recovery by a custom strategy, albeit written in Python and thus not expressed formally. PEGs had been known for not having a good error recovery algorithm, but using “labeled failures”, it is now possible to overcome this problem [49]. Even more, automatic error recovery has been designed for PEG parsing [50], but again it leaves little space for customization by the parser writer. Likewise, it is possible to perform automatic and language-independent error recovery in generalized LR parsing [51].

Note, however, that all these approaches are either based on augmenting the syntax describing formalism, e.g., with an error token, as in `yacc` and `bison`, or automatic. In either case, they are an addition to the formalism that the parser is based on and, more often than not, implemented by an auxiliary algorithm. None of these approaches supports the implementation of error recovery “within a system” (other than in a very limited form). To combat the degree of freedom in specifying error recovery, some approaches are based on designing sophisticated patterns used for describing error recovery [52].

As shown in Table 1, LLR-systems and Markov normal algorithms are the only Turing-complete formalisms that have been or are being proposed for parsing programming and domain-specific languages. Unlike other formalisms in Table 1, they can parse any language construct no matter the complexity of the language syntax. However, Markov normal algorithms were never used for parsing because they were considered hard to design and slow to run [53]. Furthermore, no approach other than the LLR-system can implement the error-recovery within a system, e.g., using productions of a context-free or parsing expression grammar.

3. Longest-Leftmost Rewriting System

3.1. Notions and Notation

An alphabet Σ is a finite set of symbols. A string w is a finite sequence of symbols from Σ. Sets Σ^* and Σ^+ contain all strings over Σ and all strings over Σ except the empty string, respectively. An empty string is denoted by ε and the length of a string w is denoted by $|w|$. Expression $k\colon w$, where $k \geq 0$ and $w \in \Sigma^*$, denotes the prefix w consisting of the first k symbols of w (or the entire w if $|w| \leq k$). A language L over Σ is a set of finite strings over Σ, i.e., $L \subseteq \Sigma^*$.

3.2. Rewriting Systems

A *rewriting (or semi-Thue) system* is a pair $\mathcal{G} = \langle V, R \rangle$, where V is a finite set of symbols and R is a finite binary relation on V^* [15]. A pair $(\alpha, \beta) \in R$ is called a *rule* and is written as $\alpha \longrightarrow \beta$. A string $\varpi \in V^*$ is called a sentential form.

A sentential form ϖ_1 *derives* another sentential form ϖ_2 using rule $r = \alpha \longrightarrow \beta$ if and only if $\varpi_1 \Longrightarrow^r_{\mathcal{G}} \varpi_2$, where a relation $\Longrightarrow^r_{\mathcal{G}}$ is defined as

$$\Longrightarrow^r_{\mathcal{G}} = \{(\omega_1 \alpha\, \omega_2, \omega_1 \beta\, \omega_2);\ \omega_1, \omega_2 \in V^*\}.$$

Likewise, ϖ_1 derives ϖ_2 using a rule string $\pi \in R^*$ if and only if $\varpi_1 \Longrightarrow^\pi_{\mathcal{G}} \varpi_2$, where a relation $\Longrightarrow^\pi_{\mathcal{G}}$ is defined inductively as

$$\begin{aligned}\Longrightarrow^\varepsilon_{\mathcal{G}} &= \mathrm{id}_{V^*}\\ \Longrightarrow^\pi_{\mathcal{G}} &= \Longrightarrow^r_{\mathcal{G}} \cdot \Longrightarrow^{\pi'}_{\mathcal{G}}\end{aligned}$$

where $\pi = r\pi'$ for some $r \in R$ and $\pi' \in R^*$. In the above definition, id_A denotes the identity relation $\{(a,a);\ a \in A\}$ and operator $\cdot$ denotes the (relational) product of relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$, defined as

$$R_1 \cdot R_2 = \{(a,c);\ a\, R_1\, b \wedge b\, R_2\, c \text{ for some } b \in B\}$$

for some sets A, B and C [15].

Furthermore, $\Longrightarrow^*_{\mathcal{G}}$ denotes $\Longrightarrow^\pi_{\mathcal{G}}$ for any $\pi \in R^*$. The subscript denoting the actual rewriting system in the names of these relations, e.g., $\mathcal{G}$, can be omitted whenever the rewriting system can be deduced from the context.

Formal grammars are special cases of rewriting systems. Let N and T be finite and disjoint sets of nonterminal and terminal symbols, respectively, and let $V = N \cup T$. A grammar G is defined as $G = \langle N, T, P, S \rangle$, where $S \in N$ is the start symbol and $P \subset V^+ \times V^*$ is a finite set of productions $\alpha \longrightarrow \beta \in P$. If $\alpha \in N$, the grammar is said to be context-free. If $0 < |\alpha| \leq |\beta|$, the grammar is context-sensitive. Sometimes these grammars are called Type 1 monotonic grammars, while the term context sensitive grammars is reserved strictly for grammars consisting of productions, where exactly one nonterminal symbol on the left side is substituted by a nonempty string of symbols [17]. However, because each context-sensitive grammar in the stricter sense can be transformed into a Type 1 monotonic grammar [19], the former are considered just a normal form of the latter and no distinction needs to be made for the purpose of this paper. If $\alpha \in N$ and $\beta \in V^*$, the grammar is said to be context-free. For a context-free grammar $\mathcal{G}$, the standard definitions of functions $\mathrm{FIRST}_k^{\mathcal{G}}$ and $\mathrm{FOLLOW}_k^{\mathcal{G}}$ are assumed [15,18,19].

Many models for describing various classes of formal languages are special cases of the rewriting system. The list of these special cases includes "generators" such as context-free or context-sensitive grammars, on the one hand, and "recognisers" such as LL and LR parsers, on the other hand. Note that both, namely generators and recognizers, can be considered as models describing a particular language, but the distinction between these two groups resembles the way they are typically used in practice. However, unlike many of its special cases, the rewriting system is Turing-complete and is thus far more general then just being either a language generator or a language recogniser.

3.3. Longest-Leftmost Rewriting System

Being so general, the rewriting system as defined above permits nondeterminism and therefore it is not very efficient if used as a language recognizer. To alleviate the efficiency issues to a significant degree, the *longest-leftmost rewriting system* (LLR-system), a special case of the rewriting system, is defined as

$$\mathcal{R} = \langle V, T, R, S, [\![,]\!] \rangle,$$

where V is a finite set of symbols, $T \subseteq V$ is a set of terminal symbols, $R \subset V^+ \times V^*$ is a set of rules, $S \in V \setminus T$ is the goal, and $[\![,]\!] \in V \setminus T$ are left and right markers. Apart from specifying T, S, and both end markers, the LLR-system differs from the general rewriting system in the form of rules and how rules are used in derivations.

Regarding the form of rules, there are two restrictions:

1. The left sides of two distinct rules in R must be different, i.e.,
$$\alpha \longrightarrow \beta \in R \implies \left(\forall \alpha' \longrightarrow \beta' \in R \colon \alpha' \neq \alpha \vee \beta = \beta'\right).$$
2. No rule can introduce, eliminate nor move a left or right marker, i.e.,
$$\begin{aligned} &\forall \alpha \longrightarrow \beta \in R \colon \exists \alpha', \beta' \in V \setminus \{[\![,]\!]\} \colon \\ &\quad ((\alpha \longrightarrow \beta = \alpha' \longrightarrow \beta') \vee (\alpha \longrightarrow \beta = [\![\alpha']\!] \longrightarrow [\![\beta']\!]) \vee \\ &\quad (\alpha \longrightarrow \beta = [\![\alpha' \longrightarrow [\![\beta') \vee (\alpha \longrightarrow \beta = \alpha']\!] \longrightarrow \beta']\!])). \end{aligned}$$

Regarding the usage of rules in derivations, the LLR-system enforces the *longest-leftmost* principle which states that, when considering a sentential form at each step of a derivation, the longest of its leftmost substrings constituting the left side of some rule in R is replaced by the right side of that rule. Therefore, a single step of using a rule $r = \alpha \longrightarrow \beta \in R$ in a derivation is described by a relation $(\Longrightarrow_{\mathcal{R}}^{r}) \subseteq V^* \times V^*$, where

$$\omega_1 \alpha \omega_2 \Longrightarrow_{\mathcal{R}}^{r} \omega_1 \beta \omega_2$$

if and only if

1. $\omega_1, \omega_2 \in V^*$ (as in a general rewriting system) and
2. The condition

$$\omega_1'\alpha'\omega_2' = \omega_1\alpha\,\omega_2 \implies |\omega_1| < |\omega_1'| \vee (|\omega_1| = |\omega_1'| \wedge |\alpha| \geq |\alpha'|)$$

holds for all $\omega_1', \omega_2' \in V^*$ and $\alpha' \longrightarrow \beta' \in R$ (as required by the longest-leftmost principle).

The combination of (a) the first restriction on the form of rules, which ensures distinct left sides of rules in R, and (b) the longest-leftmost principle for selecting the next rule makes the LLR-system deterministic. The second restriction on the form of rules, which preserves the position of the left and the right marker, simplifies things in a similar way, as does augmenting the context-free grammar with \$.

The language accepted by the LLR-system is defined as

$$L(\mathcal{R}) = \{\varpi \in T^*;\ \exists \pi \in R^+ \colon [\![\,\varpi]\!] \Longrightarrow^{\pi}_{\mathcal{R}} [\![S]\!]\}.$$

Hence, a parsing process is the longest possible derivation

$$[\![w]\!] = \varpi_0 \Longrightarrow^{r_1}_{\mathcal{R}} \varpi_1 \Longrightarrow^{r_2}_{\mathcal{R}} \varpi_2 \Longrightarrow^{r_3}_{\mathcal{R}} \ \ldots\ \Longrightarrow^{r_n}_{\mathcal{R}} \varpi_n,$$

where $w = \varpi$ is the input string and ϖ_n is the result. If $\varpi_n = [\![S]\!]$, parsing is successful, i.e., $w \in L(\mathcal{R})$, and $\pi = r_1 r_2 \ldots r_n$ represents the *parse* of w in regard to $\mathcal{R}$.

Example 1. *Consider an LLR-system for language* $\{a^n b^n c^n d^n;\ n \geq 0\}$ *with the following rules:*

$$\begin{array}{lllll} [\![\,a \longrightarrow [\![\,A & Aa \longrightarrow aA & Bb \longrightarrow bB & Cc \longrightarrow cC & \\ & Ab \longrightarrow B & Bc \longrightarrow C & Cd \longrightarrow \varepsilon & [\![\,]\!] \longrightarrow [\![\,S\,]\!] \end{array}$$

Parsing of string aabbccdd proceeds as follows (symbols that represent the left side of a rule to be applied at each step are underlined):

$$\begin{aligned} &\underline{[\![\,a}abbccdd\,]\!] \Longrightarrow [\![\,\underline{Aa}bbccdd\,]\!] \Longrightarrow \underline{[\![\,a}Abbccdd\,]\!] \Longrightarrow [\![\,A\underline{Ab}bccdd\,]\!] \Longrightarrow \\ &\quad \Longrightarrow [\![\,A\underline{Bb}ccdd\,]\!] \Longrightarrow [\![\,\underline{Ab}Bccdd\,]\!] \Longrightarrow [\![\,B\underline{Bc}cdd\,]\!] \Longrightarrow [\![\,B\underline{Cc}dd\,]\!] \Longrightarrow \\ &\quad \Longrightarrow [\![\,\underline{Bc}Cdd\,]\!] \Longrightarrow [\![\,C\underline{Cd}d\,]\!] \Longrightarrow [\![\,\underline{Cd}\,]\!] \Longrightarrow \underline{[\![\,]\!]} \Longrightarrow [\![\,S\,]\!] \end{aligned}$$

The last rule, i.e., $[\![\,]\!] \longrightarrow [\![\,S\,]\!]$*, extends the sentential form: the reader is invited to rewrite this* LLR*-system to an equivalent one where no rule extends the sentential form.*

Defined here for the first time, the LLR-system is meant to be used as a model of a language recognizer, i.e., parser, which transforms its input into a single symbol. At each step, it transforms the current sentential form into the next one: in step i, it transforms ϖ_{i-1} into ϖ_i using some rule $r_i \in R$ chosen according to the longest-leftmost principle.

3.4. Similar Models

A *context-sensitive reduction system* [8] is a special case of the LLR-system. It differs from the LLR-system by imposing another restriction on the form of rules: the length of the rules' right side cannot not exceed the length of its left side. As the rules of the context-sensitive reduction system never make the sentential form being processed longer, they are called reductions and the system itself is called a reduction system.

Switching the sides of the additional restriction yields the restriction on productions of context-sensitive grammars. Hence, the context-sensitive reduction system can be used to recognize deterministic context-sensitive languages even though in practice it has so far been used to parse context-free languages [13,14]. However, by avoiding the additional restriction, the LLR-system offers a more natural implementation of parsers, both traditionally table driven or manually written.

Another model similar to the LLR-system is a *Markov normal algorithm* [6]. It has been defined for studying various problems regarding computability [7,54,55] and is very similar to the LLR-system, except that the next rule is not chosen as required by the longest-leftmost principle. Instead, rules are indexed and a rule with a lower index takes precedence over all rules with higher indices regardless of where in a sentential form a rule can be applied.

Still, if a sentential form contains several appearances of the left side of a rule, the leftmost one is selected and subsequently replaced by the rule's right side. Furthermore, a rule designated as terminal ends a derivation that could have been extended further, had such a rule not have been terminal.

Both the LLR-system and the Markov normal algorithm are Turing-complete rewriting systems (see Theorem A1 in Appendix B for the former and [7] for the latter) and thus equivalent. However, to a person trained in formal grammars, the LLR-system might feel like a more natural model than the Markov algorithm with its indexed set of rules. Furthermore, the latter's formulation "*makes it difficult to specialize subclasses of Markov normal algorithm's performing particular tasks*" [53] and, despite some efficiency improvement [56], it has been noted that the implementation of Markov normal algorithms "*is intrinsically slow*" [53].

3.5. Implementation

Even though the LLR-system is fully deterministic, once efficiency issues are considered, its implementation is not as straightforward as one would assume or wish. There are two main issues to be resolved:

1. How to determine:
   (a) Which rule must be applied to a given sentential form;
   (b) Where in the sentential form it must be applied efficiently? (As it turns out, these two questions are inseparable and therefore considered a single issue.)
2. How to represent the sentential form so that both the selection and application of the next rule can be performed as efficiently as possible?

Regarding the first issue, i.e., selection and application of a rule at each step, the problem is as follows : given a set of nonempty strings over V, i.e., the left sides of all rules in $\mathcal{R}$, and a sentential form ϖ, find such strings $\omega_1, \omega_2 \in V^*$ and rule $r = \alpha \longrightarrow \beta \in R$ that

$$\varpi = \omega_1 \alpha\, \omega_2 \Longrightarrow^{r}_{\mathcal{R}} \omega_1 \beta\, \omega_2.$$

By the definition of $\Longrightarrow^{r}_{\mathcal{R}}$, there should be no other strings $\omega_1', \omega_2' \in V^*$ and no other rule $r = \alpha' \longrightarrow \beta' \in R$ so that $\varpi = \omega_1' \alpha'\, \omega_2'$, and $|\omega_1'| < |\omega_1| \vee (|\omega_1'| = |\omega_1| \wedge |\alpha'| \geq |\alpha|)$ would hold.

As the problem of selecting the next rule and finding the position of its application does not depend on the length of rules' right sides, it is exactly the same as if the context-sensitive reduction system is considered. Since the problem is the same, so are the solutions [8]:

1. *Full backward jumping algorithm:*
   The naive approach is to start at the beginning of the current sentential form $\varpi = X_1 X_2 X_3 \ldots$: if there are some left sides of rules in R that are prefixes of $X_1 X_2 X_3 \ldots$, then the longest one is selected and it must be applied at position 1 ($\omega_1 = \varepsilon$); otherwise, if there are some left sides that are prefixes of $X_2 X_3 X_4 \ldots$, then the longest one is selected, and it must be applied at position 2 ($\omega_1 = X_1$), etc. If none is found, the parsing is over.
   This is performed at each step of parsing independently, without any information about the part of the sentential form already scanned being carried from one step to another; at each step, the algorithm jumps back all the way to the beginning of the (transformed) sentential form.
2. *Limited backward jumping algorithm:*
   If the rule $\alpha \longrightarrow \beta$ has been applied in the previous step at position i, rule $\alpha' \longrightarrow \beta'$ cannot be applied in the next step left of position $(i - (|\alpha'| - 1))$, as otherwise, the rule $\alpha' \longrightarrow \beta'$ should have taken precedence over $\alpha \longrightarrow \beta$ in the previous step. Even more, depending on the overlapping of β and α', the first position, where $\alpha' \longrightarrow \beta'$ is applicable in the next step, might be even further to the right (sometimes so much that it is actually a jump forward). Hence, instead of jumping back to the beginning of

ϖ, the algorithm must jump back only as far as the rule with the longest backward jump relative to β requires (see [8] for full details).
This approach is implemented in WEB and CWEB [13,14], where the parsers are implemented by hand. A Deterministic Finite Automaton (DFA) for finding out which left side matches at the given position is implemented by a series of nested *case* (WEB is written in Pascal) and *switch* (CWEB is written in C) statements. Although error prone, the length of a backward jump for each right side is calculated by hand.
A very similar technique is used in optimizing the implementation of Markov normal algorithms [56]. However, because the indexation of rules introduces precedence, the lengths of rules already applied are kept on the auxiliary stack, which makes implementation slightly more complicated.

3. *DFA-based algorithm:*
The main idea is to keep states that a DFA, which is used to determine what rule and where, is to be applied to at each step and passed for reuse in the subsequent steps. If a DFA was used to determine that a rule $\alpha \longrightarrow \beta$ should be applied to a sentential form $\varpi = \omega_1 \alpha \omega_2$, the states that the DFA passed when scanning ω_1 should be stored for later use. Once the rule has been applied, i.e., when the sentential form is transformed into $\omega_1 \beta \omega_2$, the DFA can immediately continue scanning $\beta \omega_2$ starting from the state it reached after reading ω_1.
The appropriate DFA can be generated using the same procedure as for the context-sensitive reduction system [8]; alternatively, but with less insight into the meaning of particular states, the algorithm for transforming a regular expression into a POSIX DFA can be used as well [57].

As described in [8], the states of the DFA, which is used for finding the longest leftmost left sides of rules in the sentential form being parsed, are modeled by sets of *items*. An item $[\alpha_1 \cdot \alpha_2 + \eta]$, where $\alpha_1\alpha_2$ is the left side of some rule in R and $\alpha_2 = \varepsilon \vee \eta = \varepsilon$, in state q denotes two possibilities. If $\alpha_2 \neq \varepsilon$, then α_1 has just been seen, but α_2 are still to be seen. Otherwise, if $\alpha_2 = \varepsilon$, then $\alpha_1\eta$ has just been seen and $\alpha_1\alpha_2$ can be selected to be replaced by a rule if and only if there is no other item that could provide, by reading a few more symbols, another left side that starts earlier or is longer. Check [8] to see how items start canceling each one out when one item cannot ever trigger a reduction for being overshadowed by another.

Example 2. *To observe the difference between the three algorithms used for determining which rule is to be used at the next step and where, consider another* LLR*-system*

$$\begin{array}{cccc} bc \longrightarrow BC & bB \longrightarrow Bb & Cc \longrightarrow cC & \\ & aB \longrightarrow \varepsilon & Cd \longrightarrow \varepsilon & [\![\,]\!] \longrightarrow [\![S]\!] \end{array}$$

for language $\{a^n b^n c^n d^n;\ n \geq 0\}$. The parsing of string aabbccdd is shown in Figure 1.

Full backward jumping is simple to understand, but limited backward jumping deserves a bit of an explanation. After the first rule, i.e., bc $\longrightarrow$ BC, has been applied, the sentential form changes to $[\![aabBCcdd]\!]$. No rule can be applied to the prefix $[\![aab$—if this was possible, that rule should have been reduced first (the leftmost component of the longest leftmost principle). However, when the parser is being made, all possible contexts of the newly inserted right side BC must be considered (the dot represents a wildcard standing for any symbol):

the right side of	$b\,c \longrightarrow B\,C$	$.\ B\ C\ .\ .$	Δ
the left side of	$b\,c \longrightarrow B\,C$	$.\ .\ .\ b\ c$	$+2$
the left side of	$b\,B \longrightarrow B\,b$	$b\ B\ .\ .\ .$	-1
the left side of	$a\,B \longrightarrow \varepsilon$	$a\ B\ .\ .\ .$	-1
the left side of	$C\,c \longrightarrow c\,C$	$.\ .\ C\ c\ .$	$+1$
the left side of	$D\,d \longrightarrow \varepsilon$	$.\ .\ C\ d\ .$	$+1$
the left side of	$[\![\,]\!] \longrightarrow [\![S]\!]$	$.\ .\ .\ [\![\ ]\!]$	$+2$
			$min = -1$

Hence, as the minimal Δ for $b\,c \longrightarrow B\,C$ is -1, the next rule can only be applied at a position that is 1 place left of B (of BC just inserted) or, if this is not possible, somewhere further to the right. All subsequent steps follow the same reasoning but the minimal Δ must be computed for each rule separately.

The DFA-based algorithm uses the reduction automaton shown in Figure 2. In the initial state, q_0, there are items denoting the starting points of all left sides of rules in R. In every other state, there is also a possibility that any left side starts right there. In this simple example, no item canceling happens—see [8] to observe this.

Note, however, that there exist cases when the limited backjumping outperforms the DFA-based approach: parsing the only string of language $\{\varepsilon\}$ using the LLR*-system containing a single rule $[\![\ ]\!] \longrightarrow [\![\ S\ ]\!]$ requires two symbol operations with the former approach but five with the latter. An* LLR*-symbol consisting of rules $[\![\ a \longrightarrow [\![$ and $[\![\ ]\!] \longrightarrow [\![\ S\ ]\!]$ represents another example.*

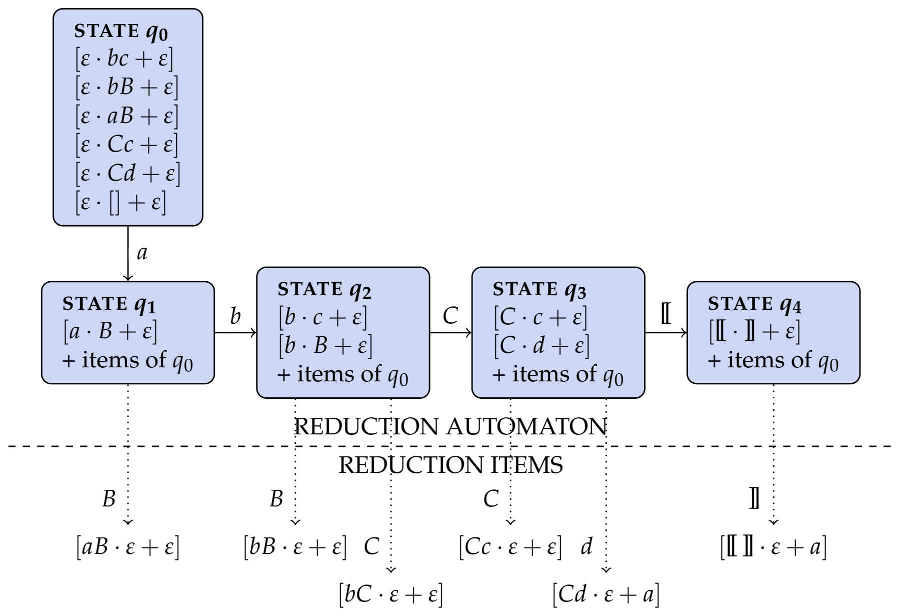
The second issue, i.e., how to represent the sentential form efficiently, is important because a rule can be applied anywhere in the sentential form and can expand or contract it. In general, a double-linked list works fine, but there are two alternatives:

1. *If the* LLR*-system is used to emulate known context-free parsing algorithms (see Section 4), which in one way or another use a stack, then it is wise to use an array for representing the sentential form.*
2. *If the* LLR*-system is used to parse very long programs of a language where it must traverse the sentential form from the left and right many times in order to apply rules at big distances (consider parsing string $a^n b^n c^n d^n$ for $n = 10^{10}$ using the* LLR*-system from Example 2), random access lists might be preferred [58].*

Once these two issues are resolved, the implementation of the LLR*-system is much the same as the implementation of the context-sensitive reduction system [8].*

	Full backward		Limited backward		DFA-based		
	$[\![\ a\,a\,b\,b\,c\,c\,d\,d\ ]\!]$	10 *cmps*	$[\![\ a\,a\,b\,b\,c\,c\,d\,d\ ]\!]$	10 *cmps*	$[\![\ a\,a\,b\,b\,c\,c\,d\,d\ ]\!]$	0 4 1 1 2 2 *	6 *cmps*
by rule $bc \longrightarrow BC$							
	$[\![\ a\,a\,b\,B\,C\,c\,d\,d\ ]\!]$	8 *cmps*	$[\![\ a\,a\,b\,B\,C\,c\,d\,d\ ]\!]$	2 *cmps*	$[\![\ a\,a\,b\,B\,C\,c\,d\,d\ ]\!]$	0 4 1 1 2 *	1 *cmp*
by rule $bB \longrightarrow Bb$							
	$[\![\ a\,a\,B\,b\,C\,c\,d\,d\ ]\!]$	6 *cmps*	$[\![\ a\,a\,B\,b\,C\,c\,d\,d\ ]\!]$	2 *cmps*	$[\![\ a\,a\,B\,b\,C\,c\,d\,d\ ]\!]$	0 4 1 1 *	1 *cmp*
by rule $aB \longrightarrow \varepsilon$							
	$[\![\ a\,b\,C\,c\,d\,d\ ]\!]$	8 *cmps*	$[\![\ a\,b\,C\,c\,d\,d\ ]\!]$	6 *cmps*	$[\![\ a\,b\,C\,c\,d\,d\ ]\!]$	0 4 1 2 3 *	3 *cmps*
by rule $Cc \longrightarrow cC$							
	$[\![\ a\,b\,c\,C\,d\,d\ ]\!]$	6 *cmps*	$[\![\ a\,b\,c\,C\,d\,d\ ]\!]$	2 *cmps*	$[\![\ a\,b\,c\,C\,d\,d\ ]\!]$	0 4 1 2 *	1 *cmp*
by rule $bc \longrightarrow BC$							
	$[\![\ a\,B\,C\,C\,d\,d\ ]\!]$	4 *cmps*	$[\![\ a\,B\,C\,C\,d\,d\ ]\!]$	2 *cmps*	$[\![\ a\,B\,C\,C\,d\,d\ ]\!]$	0 4 1 *	1 *cmp*
by rule $aB \longrightarrow \varepsilon$							
	$[\![\ C\,C\,d\,d\ ]\!]$	6 *cmps*	$[\![\ C\,C\,d\,d\ ]\!]$	6 *cmps*	$[\![\ C\,C\,d\,d\ ]\!]$	0 4 3 3 *	3 *cmps*
by rule $Cd \longrightarrow \varepsilon$							
	$[\![\ C\,d\ ]\!]$	4 *cmps*	$[\![\ C\,d\ ]\!]$	2 *cmps*	$[\![\ C\,d\ ]\!]$	0 4 3 *	1 *cmp*
by rule $Cd \longrightarrow \varepsilon$							
	$[\![\ ]\!]$	2 *cmps*	$[\![\ ]\!]$	2 *cmps*	$[\![\ ]\!]$	0 4 *	1 *cmp*
by rule $[\![\ ]\!] \longrightarrow [\![\ S\ ]\!]$							
	$[\![\ S\ ]\!]$	4 *cmps*	$[\![\ S\ ]\!]$	0 *cmps*	$[\![\ S\ ]\!]$	0 4 0 0	3 *cmps*
		58 *cmps*		34 *cmps*			21 *cmps*

Figure 1. Comparison of parsing string *aabbccdd* using the LLR-system defined in Example 2 (symbols checked and states entered at each step of parsing are typeset in red and underlined, while states typeset in green were inherited from the previous step; the asterisk denotes the position where the DFA recognized the left side of a rule).



STATE q_0
$[\varepsilon \cdot bc + \varepsilon]$
$[\varepsilon \cdot bB + \varepsilon]$
$[\varepsilon \cdot aB + \varepsilon]$
$[\varepsilon \cdot Cc + \varepsilon]$
$[\varepsilon \cdot Cd + \varepsilon]$
$[\varepsilon \cdot [] + \varepsilon]$
a
STATE q_1
$[a \cdot B + \varepsilon]$
+ items of q_0
b
STATE q_2
$[b \cdot c + \varepsilon]$
$[b \cdot B + \varepsilon]$
+ items of q_0
C
STATE q_3
$[C \cdot c + \varepsilon]$
$[C \cdot d + \varepsilon]$
+ items of q_0
$\llbracket$
STATE q_4
$[\llbracket \cdot \rrbracket + \varepsilon]$
+ items of q_0
REDUCTION AUTOMATON
REDUCTION ITEMS
B
$[aB \cdot \varepsilon + \varepsilon]$
B
$[bB \cdot \varepsilon + \varepsilon]$
C
$[bC \cdot \varepsilon + \varepsilon]$
C
$[Cc \cdot \varepsilon + \varepsilon]$
d
$[Cd \cdot \varepsilon + a]$
$\rrbracket$
$[\llbracket \rrbracket \cdot \varepsilon + a]$

Figure 2. The DFA used by the LLR-system defined in Example 2 (the missing transitions on a, b, C and $\llbracket$ lead to states 1, 2, 3 and 4, respectively; all other missing transitions lead to state 0).

4. Parsing Context-Free Languages

As the LLR-system is Turing-complete, there is no question about what can or what can not be achieved by it. Hence, the more pressing question, by far, is how easy it is to implement a language recognizer, i.e., parser, using an LLR-system for, say, present and future programming and domain-specific languages. To answer this question, it is worth looking at how it relates to other known parsing algorithms first. As the LLR-system is defined in this paper for the first time, the formulations presented in this section should serve as the initial pool of patterns helping someone to start using LLR-systems for parsing. Note that most of them are not applicable even to the context-sensitive reduction system [8], because they depend heavily on being possible to extend the current sentential form that is being parsed.

Be aware that the exposition starts with trivial formulations of existing parsing algorithms. However, this triviality is the point: it makes the LLR-system easier to understand and use for someone trained in mainstream parsing algorithms. Furthermore, the examples in this section are essential, as they are meant to illustrate, beyond the formal schemes provided, how a parser can be constructed.

4.1. Top-Down Parsing: SLL and LL

The simplest and by far the most widely used top-down parsing method is SLL(1) parsing [15]. It represents the theoretical model for handwritten recursive-descent parsers, which are (usually slightly augmented with a trick or two) built into many of the most widely used compilers nowadays, e.g., GCC and Clang/LLVM implementations of C and C++ compilers.

An SLL(k) parser for grammar $\mathcal{G} = \langle V, T, P, S \rangle \in \text{SLL}(k)$, where the $T \subset V$, as in [15], can be implemented as an LLR-system $\mathcal{R} = \langle \overline{V}, T, R, S, \llbracket, \rrbracket \rangle$ with a set of rule R, defined as

$$\begin{aligned}
\textit{startup rules: } & \llbracket\, a \longrightarrow \llbracket\, S\overline{S}a \in R && \forall a \in T \cup \{\rrbracket\} \\
\textit{shift rules: } & \overline{a}a \longrightarrow \varepsilon \in R && \forall a \in T \\
\textit{produce rules: } & \overline{A}x \longrightarrow \overline{X}_n\overline{X}_{n-1}\ldots\overline{X}_1 x \in R \\
& \forall A \longrightarrow X_1X_2\ldots X_n \in R \,\wedge \\
& \forall x \in \text{FIRST}_k^{\mathcal{G}'}(X_1X_2\ldots X_n\text{FOLLOW}_k^{\mathcal{G}'}(A)),
\end{aligned}$$

where $\overline{V} = \{S, \llbracket, \rrbracket\} \cup \{\overline{X};\ X \in V\} \cup T$ and $\llbracket, \rrbracket \notin V$; $\mathcal{G}'$ is $\rrbracket$-augmented grammar $\mathcal{G}$, i.e., using $\rrbracket$ as the "end-of-file" marker. The idea behind this scheme is simple:

1. During parsing, the sentential form at each step has the form $[\![S\overline{\delta}w]\!]$, where $\overline{\delta}$, a string consisting of the "overlined" symbols, represents the LL stack, and w, a string consisting of the not-"overlined" symbols in T, represents the remaining input.
2. At the beginning, one of the startup rules inserts S, which ensures that no startup rule can be applied again, and $\overline{S}$, which represents the initial stack contents.
3. Once the stack has been set up, shift and produce rules are applied just as they would be applied by an SLL(k) parser. If and only if the input belongs to $L(\mathcal{G})$, the stack contents and the remaining input are eliminated and the accepting sentential form $[\![\, S \,]\!]$ remains.

Example 3 (Dealing with the right recursion)**.** *As the scheme for* SLL(k) *parsers is so straightforward, an example instead of a rigorous proof, e.g., the induction on the length of a derivation, should suffice. Consider a fragment of a programming language grammar described by EBNF productions*

$$\begin{aligned} S &\longrightarrow \mathrm{id} \;=\; E \\ S &\longrightarrow \mathrm{begin}\; S \;\{\; ;\, S \;\}\; \mathrm{end} \end{aligned}$$

where E is considered a terminal for a moment. To rewrite the above grammar into an LLR*-system,* $S\ \{\ ;S\ \}$ *describing a sequence of statement is replaced by a new nonterminal L. Thus, the following startup and produce rules are obtained first (the shift rules are trivial and thus not listed here):*

$$\begin{aligned} S &\longrightarrow \mathrm{id} \;=\; E \\ S &\longrightarrow \mathrm{begin}\; L\; \mathrm{end} \end{aligned} \quad\Longrightarrow\quad \begin{aligned} [\![\, x &\longrightarrow [\![\, S\, \overline{S}\, x \\ \overline{S}\, \mathrm{id} &\longrightarrow \overline{E}\, \overline{=}\, \overline{\mathrm{id}}\, \mathrm{id} \\ \overline{S}\, \mathrm{begin} &\longrightarrow \overline{\mathrm{end}}\, \overline{L}\, \overline{\mathrm{begin}}\, \mathrm{begin} \end{aligned}$$

The first rule is actually a set of rules, where x stands for any terminal of this little language. The sequence of statements is treated in the most standard way, namely

$$\begin{aligned} L &\longrightarrow S\, L' \\ L' &\longrightarrow \varepsilon \\ L' &\longrightarrow ;\, S\, L' \end{aligned} \quad\Longrightarrow\quad \begin{aligned} \overline{L}\, x &\longrightarrow \overline{L'}\, \overline{S}\, x \\ \overline{L'}\, \mathrm{end} &\longrightarrow \mathrm{end} \\ \overline{L'}\, ; &\longrightarrow \overline{\mathrm{L}'}\, \overline{\mathrm{S}}\, \overline{;}\, ; \end{aligned}$$

where $x \in \mathrm{FIRST}(SL'\mathrm{FOLLOW}(L)) = \{\mathrm{id}, \mathrm{begin}, \mathrm{if}\}$ *(as anyone familiar with* SLL *parsing knows, x could also be left out entirely).*

To see how parsing works, consider the first few steps of the parsing string begin id $=$ E; id $=$ E end*:*

$$\begin{aligned} & [\![\, \mathrm{begin\,id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \\ \Longrightarrow\; & [\![\, \mathrm{S}\, \overline{S}\, \mathrm{begin\,id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \\ \Longrightarrow\; & [\![\, \mathrm{S}\, \overline{\mathrm{end}}\, \overline{L}\, \overline{\mathrm{begin}}\, \mathrm{begin\,id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \\ \Longrightarrow\; & [\![\, \mathrm{S}\, \overline{\mathrm{end}}\, \overline{L}\, \mathrm{id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \\ \Longrightarrow\; & [\![\, \mathrm{S}\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{S}\, \mathrm{id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \\ \Longrightarrow\; & [\![\, \mathrm{S}\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{E}\, \overline{=}\, \overline{\mathrm{id}}\, \mathrm{id} \;=\; E\, ;\mathrm{id} \;=\; E\, \mathrm{end} \,]\!] \end{aligned}$$

Hence, after the initial S, which waits to announce success once the parsing is over, the "overlined" symbols represent the stack (with the topmost symbol on the right) and the rest is the yet unparsed part of the input string.

Example 4 (The dangling else problem)**.** *Suppose that the* if*-statement defined by EBNF production*

$$S \longrightarrow \mathrm{if}\; E\; \mathrm{then}\; S\; [\; \mathrm{else}\; S\;]$$

is added to the grammar introduced in Example 3. To deal with the dangling else *problem, the ambiguity of a language must first be eliminated. Assuming the standard syntax of the* if*-statement, i.e., the* else*-branch is a part of the most recent preceding* else*-free* if*-statement, the EBNF production can be transformed into a set of productions resulting in a unambiguous but bloated grammar [1,18]. Alternatively, it can be transformed elegantly into rules*

$$\begin{aligned} S &\longrightarrow \mathrm{if}\; E\; \mathrm{then}\; S \\ S &\longrightarrow \mathrm{if}\; E\; \mathrm{then}\; S\; \mathrm{else}\; S \end{aligned} \quad\Longrightarrow\quad \begin{aligned} \overline{S}\, \mathrm{if} &\longrightarrow \overline{S}\, \overline{\mathrm{else}}\, \overline{S}\, \overline{\mathrm{then}}\, \overline{E}\, \overline{\mathrm{if}}\, \mathrm{if} \\ \overline{S}\, \overline{\mathrm{else}}\, x &\longrightarrow x \\ \overline{\mathrm{else}}\, \mathrm{else} &\longrightarrow \varepsilon \end{aligned}$$

where $x \in \mathrm{FOLLOW}(S) \setminus \{\mathrm{else}\} = \{\mathrm{end}, ;, \rrbracket\}$ *or simply* $x \neq \mathrm{else}$. *Namely, the produce rule ignores the dangling* else *problem, which is then fixed by the second rule (the third rule is just the regular shift rule). To be sure, these three rules, which describe how the* if*-statement above is handled, are exactly how a recursive function responsible for parsing the* if*-statement is usually written: once the statement in the* then*-branch has been parsed, it checks whether there is a matching* else*—if there is none, it skips the* else*-branch. Note, however, that the decision made during the production is being parsed.*

Note, however, that the LLR*-system makes it possible to express this with just another rule (albeit neither a shift nor produce); within a system and without a hack, e.g., choosing a shift over reduce by default to resolve a conflict, like most* LALR *parser generators do.*

Example 5 (Extending the lookahead)**.** *Suppose that the EBNF production describing the assignment statement of grammar in Example 3 is modified into*

$$S \longrightarrow \mathrm{id}\ \hat{}?\ =\ \mathrm{E}$$

that can be replaced by productions $S \longrightarrow \mathrm{id}\ P\ =\ \mathrm{E}$ *and* $P \longrightarrow \varepsilon \mid \hat{}$, *which fulfill the* $\mathrm{SLL}(1)$ *condition. However, by extending the lookahead buffer from* 1 *to* 2 *locally, i.e., just in case of the assignment statement, the transformation*

$$\begin{array}{l} S \longrightarrow \mathrm{id}\ =\ \mathrm{E} \\ S \longrightarrow \mathrm{id}\ \hat{}\ =\ \mathrm{E} \end{array} \quad \Longrightarrow \quad \begin{array}{r} \overline{S}\,\mathrm{id} \longrightarrow \overline{E}\,\overline{=}\,\overline{\mathrm{id}}\,\mathrm{id} \\ \overline{S}\,\mathrm{id}\ \hat{} \longrightarrow \overline{E}\,\overline{=}\,\overline{\hat{}}\,\overline{\mathrm{id}}\,\mathrm{id}\ \hat{} \end{array}$$

can be used without introducing an additional nonterminal symbol and productions expanding it.

Looking two, three, or a few symbols ahead instead of just one symbol ahead is an extension used in many hand-coded recursive-descent parsers. Moreover, again, unlike grammars that are transformed into a parser by a generator, the LLR*-system makes it possible to express such a local change.*

To summarize, $\mathrm{SLL}(k)$ grammars can be transformed into an LLR-system in a straightforward manner (Example 3). If needed, the LLR-system can inspect several topmost stack symbols (Example 4) or more than k input symbols whenever needed (Example 5), or both. Finally, an LLR-system obtained by the transformation from an $\mathrm{SLL}(k)$ grammar, as defined above, is not a context-sensitive reduction system: startup rules and (in practice most of) produce rules extend the sentential form and thus violate restrictions imposed by a context-sensitive reduction system.

As the canonical $\mathrm{LL}(k)$ parser can also be described as a rewriting system with a set of rules M containing shift and produce actions [16], it can be transformed into an LLR-system in more or less the same way as the $\mathrm{SLL}(k)$ parser:

$$\begin{array}{rl} \textit{startup rules:} & \llbracket\, a \longrightarrow \llbracket\, S\, [\llbracket]\, [\llbracket S]\, a \in R\ \forall a \in T \cup \{\rrbracket\} \\ \textit{shift rules:} & [\delta a]\, ax \longrightarrow x \in R \\ & \quad\quad \text{iff } [\delta a] \bullet ax \longrightarrow \varepsilon \bullet x \in M \\ \textit{produce rules:} & [\delta]\, [\delta A]\, x \longrightarrow [\delta]\, [\delta X_n]\, [\delta X_n X_{n-1}]\ \ldots\ [\delta X_n X_{n-1} \ldots X_1]\, x \in R \\ & \quad\quad \text{iff } [\delta]\, [\delta A] \bullet x \longrightarrow [\delta]\, [\delta X_n]\, [\delta X_n X_{n-1}]\ \ldots\ [\delta X_n X_{n-1} \ldots X_1] \bullet x \in M \\ \textit{final rule:} & \llbracket\, S\, [\llbracket]\, \rrbracket \longrightarrow \llbracket\, S\, \rrbracket \in R. \end{array}$$

Remember that $[\delta]$, where $\delta \in (V \cup \{\llbracket, \rrbracket\})^*$, represents the equivalence class of viable suffixes of the $\llbracket, \rrbracket$-augmented grammar $\mathcal{G}'$ in regard to $\mathrm{LL}(k)$-equivalence [16] or, in other words, a state of the canonical $\mathrm{LL}(k)$ parser.

The idea behind these scheme is the same as for the $\mathrm{SLL}(k)$ parser, but as the canonical $\mathrm{LL}(k)$ parser needs two topmost states to decide the next produce action, the state $[\llbracket]$ is inserted beneath $[\llbracket S]$ at the beginning and therefore must be removed at the end. Again, the startup and produce rules violate restrictions of the context-sensitive reduction system. However, as the canonical $\mathrm{LL}(k)$ parser is rarely needed in practice even for $k = 1$ and as in practice it cannot be produced without a generator except for the smallest and simplest grammars, it is not considered as an important model for formulating parsers using the LLR-system.

4.2. Bottom-Up Parsing: LR and LALR

The most widely used bottom-up parsing method is LALR parsing, or more precisely, LA(1)LR(0) parsing. Even though it is usually implemented by a generated table-driven parser, in theory both the canonical LR(k) and LA(k)LR(l) parsers are formulated as rewriting systems with shift and reduce actions [16]. If described by a set of actions M, either one can be transformed into an LLR-system with a set of rules R, defined as follows:

$$\begin{aligned}
\textit{startup rules:}\ & [\![\, a \longrightarrow [\![\, S\, [[\![]\, a \in R \qquad \forall a \in T \cup \{]\!]\} \\
\textit{shift rules:}\ & [\delta]\, ax \longrightarrow [\delta]\, [\delta a]\, x \in R \\
& \text{iff } [\delta] \bullet ax \longrightarrow [\delta]\, [\delta a] \bullet x \in M \\
\textit{reduce rules:}\ & [\delta]\, [\delta X_1]\, [\delta X_1 X_2]\ \ldots\ [\delta X_1 X_2 \ldots X_n]\, x \longrightarrow [\delta]\, [\delta A]\, x \in R \\
& \text{iff } [\delta]\, [\delta X_1]\, [\delta X_1 X_2]\ \ldots\ [\delta X_1 X_2 \ldots X_n] \bullet x \longrightarrow [\delta]\, [\delta A] \bullet x \in M \\
\textit{final rule:}\ & [\![\, S\, [[\![]\, [[\![S]\,]\!] \longrightarrow [\![\, S\,]\!] \in R.
\end{aligned}$$

Remember that $[\delta]$, where $\delta \in (V \cup \{[\![,]\!]\})^*$ represents the equivalence class of viable prefixes of the $[\![,]\!]$-augmented grammar $\mathcal{G}'$ in regard to LR(k)- or LA(k)LR(l)-equivalence [16], or, in other words, a state of the canonical LR(k) of the LA(k)LL(l) parser, respectively. Again, formulating an LR or LALR parser for a full-scale programming language as an LLR-system in practice is out of the question because of the size and complexity of the underlying LR or LALR automata and consequently the number of states and rules.

Despite the complexity of LR parsing, one should embrace rather than abandon the shift-reduce parsing, as it is the most natural solution for parsing sentential forms where a left recursion is essential.

Example 6 (Essential left recursion)**.** *One of the most apparent examples of essential left recursion is the description of arithmetic expressions. Unlike the sequence of statements (as in Example 3) where both left or right recursion work equally well, the left recursive productions allows neat abstract syntax trees because they enforce left-associativity of operators [18]. Fortunately, for the well known grammar for arithmetic expressions, there exists a simple and intuitive* LLR-system *with a set of rules*

$$\begin{aligned}
E &\longrightarrow E + T \mid T \\
T &\longrightarrow T * F \mid F \\
F &\longrightarrow \text{num} \mid \text{id} \mid (\ E\ )
\end{aligned}
\qquad \Longrightarrow \qquad
\begin{aligned}
\text{num} &\longrightarrow F \\
\text{id} &\longrightarrow F \\
(\ \text{E}\ ) &\longrightarrow F \\
F &\longrightarrow T \\
T * F &\longrightarrow T \\
T\ x &\longrightarrow E\ x \\
E + T\ x &\longrightarrow E\ x
\end{aligned}$$

where x is any symbol other than $*$ (or belonging to $\{+,),]\!]\}$ if one takes the pain of finding this out manually). The first three and the fourth rule are obvious. The fifth and the sixth rules are crucial for understanding this LLR-system*:*

1. *As soon as $T * F$ is observed, it should be reduced to T. Because the leftmost rule applies first, "as soon as" (rather than just if), the left associativity of $*$ is ensured.*
2. *However, if and only if T cannot be extended any further, i.e., if it is followed by anything other than $*$, it can and thus should be reduced to E.*

Likewise, $E + T$ can be reduced to E if and only if T cannot be extended any further, i.e., if it is not followed by $*$, the operator with the higher precedence.

Parsing proceeds in similar way as in Example 3. To see why the condition imposed by x in the sixth rule is needed, consider the derivation

$$[\![\text{id}*\text{id}]\!] \Longrightarrow [\![F*\text{id}]\!] \Longrightarrow [\![T*\text{id}]\!] \Longrightarrow [\![E*\text{id}]\!] \Longrightarrow \ldots$$

without x in the sixth rule: the first id *is first reduced to E and never reduced any further.*

The above transformation can be considered as a pattern just as much as another transformation, which involves left recursion elimination and results in SLL(1) *grammar. It produces a shift-reduce parser that is much simpler than the corresponding* LR(1) *parser with 24 states or* LALR *parser with 13 states. Lastly, it can be observed as an example of operator-precedence parsing [59].*

4.3. Bidirectional Parsing: Top-Down and Bottom-Up

Sticking to a preselected strategy, either top-down or bottom-up, is not always the best choice for all parts of the language being parsed. For instance:

1. Sequences of statements or declarations can be expressed using either a left or a right recursion. However, if described by the right recursion and parsed using the top-down parser, e.g., shift-produce, each statement or declaration is appended to the abstract syntax tree as soon as it has been parsed. If the left recursion is used, it can be appended to the tree only after the entire sequence has been parsed.
2. Expressions are best described by the left recursion because, unlike the right recursion, it allows describing the left-associativity of arithmetic operators in the best possible way [18]. To avoid (a) eliminating the left-recursion when the parser is being made and (b) transforming abstract syntax trees when the parser is being run, one should use the bottom-up, e.g., shift-reduced, parser.

Bidirectional parsers, i.e., those incorporating both top-down and bottom-up parsing, have been investigated before. Most of these parsers are variants of either left-corner parsing [60–62] or a combination of LL and LR parsing [11,12]. Parsers based on these approaches have never become widely used, most likely because of their complexity, rigidity, non-intuitiveness, and perhaps because of lack of tools, as none of them can be hand-coded. However, as the following two examples show, LLR-system represents a platform for hand-coded bidirectional parsers.

Example 7 (Bottom-up parsing during top-down parsing)**.** *The statements of the little programming languages are described by an EBNF grammar in Examples 3–5. By rewriting it into an* SLL(1) *grammar first, it has been transformed into an* LLR*-system which performs a top-down parsing:*

$$\begin{array}{rcl} S & \longrightarrow & \mathrm{id} \; = \; E \\ S & \longrightarrow & \mathrm{id}\;\hat{} \; = \; \mathrm{E} \\ S & \longrightarrow & \mathrm{begin}\; L \;\mathrm{end} \\ S & \longrightarrow & \mathrm{if}\; E \;\mathrm{then}\; S \\ S & \longrightarrow & \mathrm{if}\; E \;\mathrm{then}\; S \;\mathrm{else}\; S \\ L & \longrightarrow & S\; L' \\ L' & \longrightarrow & \varepsilon \\ L' & \longrightarrow & ;\; S\; L' \end{array} \quad\Longrightarrow\quad \begin{array}{rcl} \llbracket\, x_1 & \longrightarrow & \llbracket\, S\,\overline{S}\,x_1 \\ \overline{S}\;\mathrm{id} & \longrightarrow & \overline{E}\,\overline{=}\,\overline{\mathrm{id}}\;\mathrm{id} \\ \overline{S}\;\mathrm{id}\;\hat{} & \longrightarrow & \overline{E}\,\overline{=}\,\overline{\hat{}}\,\overline{\mathrm{id}}\;\mathrm{id}\;\hat{} \\ \overline{S}\;\mathrm{begin} & \longrightarrow & \overline{\mathrm{end}}\;\overline{L}\;\overline{\mathrm{begin}}\;\mathrm{begin} \\ \overline{S}\;\mathrm{if} & \longrightarrow & \overline{S}\;\overline{\mathrm{else}}\;\overline{S}\;\overline{\mathrm{then}}\;\overline{E}\;\overline{\mathrm{if}}\;\mathrm{if} \\ \overline{S}\;\overline{\mathrm{else}}\;x_2 & \longrightarrow & x_2 \\ \overline{L}\;x_3 & \longrightarrow & \overline{L'}\;\overline{S}\;x_3 \\ \overline{L'}\;\mathrm{end} & \longrightarrow & \mathrm{end} \\ \overline{L'}\;; & \longrightarrow & \overline{\mathrm{L}'}\,\overline{\mathrm{S}}\,\overline{;}\,; \end{array}$$

where x_1, x_2 and x_3 belong to $T \cup \{\rrbracket\}$, $\{\mathrm{end}, ;, \rrbracket\}$ and $\{\mathrm{id}, \mathrm{begin}, \mathrm{if}\}$, respectively.

So far, symbol E denoting expressions has been treated as a terminal symbol (Example 3). Suppose expressions have the form as described in Example 6. By adding rules from Example 6 for parsing expressions and rules

$$\overline{\mathrm{E}}\;E\;x \longrightarrow x$$

*where x is anything but $+$ (which extends an expression as E is a left recursive symbol), and the resulting parser becomes bidirectional. Namely, sentences are parsed top-down while expressions are parsed bottom-up. Note that $\overline{\mathrm{E}}$ is never expanded by any "*SLL(1) *rule" and thus the parser slips into a bottom-up parsing of an expression. The expression is reduced into symbol E, which must then be removed together with $\overline{\mathrm{E}}$ so that top-down parsing can continue.*

To see how this works in practice, consider the parsing program "$\mathrm{if}\ \mathrm{x} + 1\ \mathrm{then}\ \mathrm{x}\ =\ 2 * \mathrm{x}$*". It starts with the top-down mode:*

$$\begin{aligned}
&\llbracket\ \text{if id} + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \text{if id} + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ \overline{\text{if}}\ \text{if id} + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ \text{id} + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket
\end{aligned}$$

With $\overline{E}$ *"at the top of the* LL *stack", the parser switches into a bottom-up mode:*

$$\begin{aligned}
&\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ \text{id} + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ F + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ T + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ E + \text{num then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ E + F\ \text{then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ E + T\ \text{then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \overline{E}\ E\ \text{then id} \ = \ \text{num} * \text{id}\ \rrbracket \\
\Longrightarrow\ &\llbracket\ S\ \overline{S}\ \overline{\text{else}}\ \overline{S}\ \overline{\text{then}}\ \text{then id} \ = \ \text{num} * \text{id}\ \rrbracket
\end{aligned}$$

By removing $\overline{\mathrm{E}}\ \mathrm{E}$ *, the parser has slipped back into the top-down mode. It will perform another pass of the bottom-up parsing to parse* num $*$ x*, upon which it will return to the top-down again.*

Example 8 (Top-down parsing during bottom-up parsing)**.** *The bidirectional parser in Example 7 starts in the top-down mode and occasionally switches to the bottom-up mode. However, it is also quite easy to switch back to a top-down mode while the parser is in a bottom-up-mode. If the small language is extended by EBNF production* $F \longrightarrow \text{id}\ (\ E\ (, E)?\ )$ *describing function calls, the following rules must be added to the* LLR*-system of Example 7:*

$$\begin{aligned}
F &\longrightarrow \text{id}\ (\ ) \\
F &\longrightarrow \text{id}\ (\ A\ ) \\
A &\longrightarrow E\ A' \\
A' &\longrightarrow \varepsilon \\
A' &\longrightarrow\ , \mathrm{E}\ \mathrm{A}'
\end{aligned}
\quad\Longrightarrow\quad
\begin{aligned}
\text{id}\ x_4 &\longrightarrow F\ x_4 \\
\text{id}\ (\ ) &\longrightarrow F \\
\text{id}\ (\ A\ ) &\longrightarrow F \\
\text{id}\ (\ x_5 &\longrightarrow \overline{\text{id}}\ (\ A\ \overline{A}\ x_5 \\
\overline{A}\ x_6 &\longrightarrow \overline{A'}\ \overline{E}\ x_6 \\
\overline{A'}\ ) &\longrightarrow\ ) \\
\overline{A'}\ , &\longrightarrow \overline{A'}\ \overline{E}\ \bar{,}\ ,
\end{aligned}$$

where $x_4 \notin \{(\}$ *and* $x_5, x_6 \in \{\text{num}, \text{id}, (\}$*. Note that when* id (x_5 *is observed, the parser switches from a bottom-up into top-down mode only to reenter the bottom-up when each expression, i.e., an argument to a function call, is expected.*

5. Error Recovery

Error detection is relatively simple, as most parsing algorithms used nowadays have the correct prefix property (stating that each recognized prefix of the input can be extended into a valid input). The hard part is error recovery. However, unlike parsing, which is one of the most formalized parts of a compiler, error recovery is not. Actual errors, i.e., those made by the programmer, are not a well-defined concept at all and any formal definition would most likely conflict with the way they are detected by a particular parser [16].

Global error recovery is about finding the correct string as similar to the actual but erroneous input string. The similarity is usually measured by Hamming distance based on the insertion, deletion, replacement or sometimes even transposition of symbols. As global error recovery is not practical during compilation, most error recovery parsers, e.g., hand-coded recursive-descent or generated by `yacc`/`bison` or ANTLR4 parser generators [5,18,26] resort to panic-mode or phrase-level error recovery. These two methods cannot be expressed by context-free grammars alone and must therefore be added to the parser based on grammar separately. As shown below, they can be incorporated into an LLR-system quite naturally. Hence, this section introduces new patterns for implementing these two error recovery techniques in an LLR-system.

5.1. Panic-Mode Error Recovery

Upon encountering a syntax error, the panic-mode error recovery skips a part of the input string until a substring belonging to a preselected set of synchronizing substrings, or *followers*, appears [18,47]. Consider, for example, an SLL(k) parser. A syntax error can be detected in two cases:

1. No produce action can be performed because (with a nonterminal as the topmost symbol) the analysis reached a sentential form $[\![\, \overline{\delta}\, \overline{A}\, z \,]\!]$, where $z \in T^*$ and $k\colon z]\!] \notin \mathrm{FIRST}_k^{\mathcal{G}'}(A\,\mathrm{FOLLOW}_k^{\mathcal{G}'}(A))$:
   Skipping over the erroneous part of the input that cannot be derived from A is described by a new symbol $\overline{A}_{\mathrm{F}}$, where F is a set of followers of A, and rules
   $$\begin{array}{lrcll} \textit{switch to panic-mode:} & \overline{A}y & \longrightarrow & \overline{A}_{\mathrm{F}}\, y & \forall y \notin \mathrm{FIRST}_{k'}^{\mathcal{G}'}(A\,\mathrm{FOLLOW}_{k'}^{\mathcal{G}'}(A)) \\ \textit{remove input symbols:} & \overline{A}_{\mathrm{F}}\, ax & \longrightarrow & \overline{A}_{\mathrm{F}}\, x & \forall ax \in T^{*k'} \setminus \mathrm{F} \\ \textit{switch back to parsing:} & \overline{A}_{\mathrm{F}}\, y & \longrightarrow & y & \forall y \in \mathrm{F} \end{array}$$
   where $k' \geq k$. The "switch to panic-mode" rules insert symbol $\overline{A}_F$, which actually performs skipping using the "remove input symbols" rules one symbol at a time until one of the "switch back to parsing" rules recognizes one of the followers and ends the skipping. In most applications $k' = k = 1$, but as extending the lookahead is not a problem in an LLR-system, the probability of a successful synchronization can be increased by choosing $k' > k$.
   Choosing the right set of synchronizing substrings depends heavily on the language being parsed: $\mathrm{F} = \mathrm{FOLLOW}_{k'}^{\mathcal{G}'}(A)$ is only the simplest case, while several authors advise choosing F in a slightly different way [16,18,47].
2. *No shift action can be performed because (with a terminal as the topmost symbol) the analysis reached a sentential form* $[\![\, \overline{\delta}\, \overline{a}\, bz \,]\!]$ *where* $a, b \in T$, $a \neq b$ *and* $z \in T^*$:
   By adding rules $\overline{a}\, b \longrightarrow b$ for all $b \neq a$, $\overline{a}$ is eliminated. The overall effect is that the missing a has just been inserted into the input string [18].

Example 9. *Consider again the language defined in Examples 3–6. The parsing of erroneous input* begin id num ; num end *starts as*

$$\begin{array}{rl} & [\![\, \mathrm{begin\,id\,num\,;num\,end} \,]\!] \\ \Longrightarrow & [\![\, S\, \overline{S}\, \mathrm{begin\,id\,num\,;num\,end} \,]\!] \\ \Longrightarrow^* & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{E}\, \overline{=}\, \mathrm{num\,;num\,end} \,]\!] \end{array}$$

when the shift action cannot be performed as symbols $\overline{=}$ *and* num *does not match. Hence, the error recovery rule* $\overline{=}\,\mathrm{num} \longrightarrow \mathrm{num}$ *is used and parsing continues as*

$$\begin{array}{rl} \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{E}\, \mathrm{num\,;num\,end} \,]\!] \\ \Longrightarrow^* & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, ;\mathrm{num\,end} \,]\!] \\ \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{S}\, \overline{;}\, ;\mathrm{num\,end} \,]\!] \\ \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{S}\, \mathrm{num\,end} \,]\!] \end{array}$$

when no produce action can be performed as no statement derived from S starts with a number. Hence, the parser switches to the error recovery mode using the "switch to panic-mode" rule $\overline{S}\,\mathrm{num} \longrightarrow \overline{S}_F\,\mathrm{num}$ *and skips the part of input it cannot parse, i.e.,* num, *until it finds a symbol in* $F = \mathrm{FOLLOW}(S) = \{;, \mathrm{else}, \mathrm{end},]\!]\}$:

$$\begin{array}{rl} \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{S}_F\, \mathrm{num\,end} \,]\!] \\ \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \overline{S}_F\, \mathrm{end} \,]\!] \\ \Longrightarrow & [\![\, S\, \overline{\mathrm{end}}\, \overline{L'}\, \mathrm{end} \,]\!] \end{array}$$

From this, parsing proceeds successfully without any further error recovery.

The above scheme of implementing a panic-mode error recovery in an LLR-system is a very simple one. If the typical recursive-descent implementation of an SLL(k) parser is considered, the error is always detected within a function implementing the analysis of phrases derived from one particular nonterminal. As it has been demonstrated that this additional information, namely which nonterminal is being parsed at the moment, can help during error recovery [5,47], the scheme of implementing an SLL(k) parser is slightly modified:

$$\begin{aligned}
\textit{startup rules: } & \llbracket\, a \longrightarrow \llbracket\, S\overline{S}a \in R && \forall a \in T \cup \{\rrbracket\} \\
\textit{shift rules: } & \overline{a}a \longrightarrow \varepsilon \in R && \forall a \in T \\
\textit{produce rules: } & \overline{A}x \longrightarrow \overline{A}_{\bullet}\overline{X}_n\overline{X}_{n-1}\ldots\overline{X}_1 x \in R \\
& \forall A \longrightarrow X_1X_2\ldots X_n \in R \,\wedge \\
& \forall x \in \mathrm{FIRST}_k^{\mathcal{G}'}(X_1X_2\ldots X_n\,\mathrm{FOLLOW}_k^{\mathcal{G}'}(A)), \\
\textit{delete rules: } & \overline{A}_{\bullet}a \longrightarrow a \in R && \forall a \in T \cup \{\rrbracket\}
\end{aligned}$$

In this scheme, each symbol $\overline{A}_{\bullet}$ represents the left side of a production that the symbols next to it were produced by. In terms of recursive-descent parsing, they represent the function responsible for parsing phrases derived from nonterminal A. Once all symbols derived from A have been parsed successfully, $\overline{A}_{\bullet}$ is simply deleted (as the function for A returns).

However, if a syntax error occurs when a symbol X_l should be parsed, symbol $\overline{A}_{\bullet}$ can be used for reducing a set of followers from all contexts, as symbol X_l can appear in just one context [5,16]. More precisely, if a syntax error is detected in $\llbracket\, \overline{\delta}\,\overline{A}_{\bullet}\,\overline{X}_n\,\overline{X}_{n-1}\ldots\overline{X}_{l+1}\,\overline{X}_l z\rrbracket$ and the substring derived from X_l cannot be parsed, the set of followers of X_l can be narrowed to $\mathrm{FIRST}_k^{\mathcal{G}'}(X_{l+1}X_{l+2}\ldots X_n\,\mathrm{FOLLOW}_k^{\mathcal{G}'}(A))$, while ignoring all other occurrences of X_l in the grammar. Furthermore, the sequence of symbols $\overline{A}_{\bullet}$ in the current sentential form, which is being parsed with the LLR-system, represents the nodes along the branch in the derivation tree leading from the root to the leaf, where the syntax error has been detected and can be used to provide the context needed for generating much more informative error messages.

Adding a panic-mode error recovery to the canonical LL(k) parser is achieved in the same way as shown above for the SLL(k) parser, while LR parsers require a slightly different approach. During the parser construction, nonterminals "representing major program pieces, such as an expression, statement or block" [18] are preselected for possible error recovery. When a syntax error is detected, the top-most states are removed from the top of the stack one by one until one of the states $[\delta]$, such that the state $[\delta A]$ exists for some preselected nonterminal A, appears on the top of the stack. With $[\delta]$ at the top of the stack, state $[\delta A]$ is pushed on the stack and the input symbols are removed until a follower of A is found. In the LLR-system, this can be described neatly with the following rules:

$$\begin{aligned}
&\textit{switch to panic-mode:} && [\delta]\,x \longrightarrow [\delta]\,[]_{\mathrm{E}}\,x && \text{no shift or reduce action on } k\!:\!x \text{ in } [\delta] \\
&\textit{remove top-most states:} && [\delta]\,[]_{\mathrm{E}} \longrightarrow []_{\mathrm{E}} && [\delta A] \text{ does not exist for any } A \in \mathrm{E} \\
&\textit{push a new state:} && [\delta]\,[]_{\mathrm{E}} \longrightarrow [\delta]\,[\delta A]\,[]_A && [\delta A] \text{ exists for some } A \in \mathrm{E} \\
&\textit{remove input symbols:} && []_A\,ax \longrightarrow []_A\,x && \forall ax \in T^{*k'} \setminus \mathrm{F}_A \\
&\textit{switch back to parsing:} && []_A\,y \longrightarrow y && \forall y \in \mathrm{F}_A
\end{aligned}$$

In this scheme, the set E contains all preselected nonterminals representing "major program pieces" while sets F_A represent the legal followers of nonterminals A, where $A \in \mathrm{E}$, of length $k' \geq k$. As before, choosing $k' > k$ increases the probability of a successful synchronization.

Example 10. *To observe how this formulation of panic-mode error recovery works, consider parsing the erroneous expression* num + (num + + num) *using the* LLR*-system based on the* LR(1) *parser for the grammar in Example 6 with the set of preselected nonterminals* $\mathrm{E} = \{E\}$*:*

$$\begin{aligned}
& \llbracket\,\mathrm{num} + (\mathrm{num} + + \mathrm{num})\,\rrbracket \\
\Longrightarrow\; & \llbracket\, S\,[\llbracket]\,\mathrm{num} + (\mathrm{num} + + \mathrm{num})\,\rrbracket \\
\Longrightarrow^*\; & \llbracket\, S\,[\llbracket]\,[\llbracket E]\,[\llbracket E+]\,[\llbracket E+(]\,[\llbracket E+(E]\,[\llbracket E+(E+]\,+\mathrm{num})\,\rrbracket
\end{aligned}$$

In state $[\![E{+}(E{+}]$, a syntax error is detected, as there is no action on $+$ and thus there is no applicable rule in R of an LLR*-system based on an* LR(1) *action. Applying the "switch to panic-mode" rule $[\![E{+}(E{+}]\,+ \longrightarrow [\![E{+}(E{+}]\,[]_{\mathrm{E}}+$, the parsing continues as*

$$\begin{aligned}
&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,[\![E{+}(E{+}] + \mathrm{num})\,]\!]\\
\Longrightarrow\;&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,[\![E{+}(E{+}]\,[]_{\mathrm{E}} + \mathrm{num})\,]\!]\\
\Longrightarrow\;&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,[]_{\mathrm{E}} + \mathrm{num})\,]\!]\\
\Longrightarrow\;&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[]_{\mathrm{E}} + \mathrm{num})\,]\!]
\end{aligned}$$

when the state $[\![E{+}(]$ appears on the top of the stack. A (new instance of) state $[\![E{+}(E]$ is pushed on the stack and the remaining part of the expression derived from (just pushed) E is removed from the input (note the difference between E *and E in the subscript):*

$$\begin{aligned}
&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[]_{\mathrm{E}} + \mathrm{num})\,]\!]\\
\Longrightarrow\;&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,[]_{E} + \mathrm{num})\,]\!]\\
\Longrightarrow^*&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,[]_{E})\,]\!]\\
\Longrightarrow\;&[\![\, S\,[\![]\,[\![E]\,[\![E{+}]\,[\![E{+}(]\,[\![E{+}(E]\,)\,]\!]
\end{aligned}$$

From this point in the derivation, the parsing can proceed without any further error recovery.

Finally, panic-mode error recovery adds a number of rules to an LLR-system, but it resembles filling all empty cells of a parse table or populating all parse functions of a recursive-descent parser with error branches.

5.2. Phrase-Level Error Recovery

The other popular method is the phrase-level error recovery. It is based on insertion, deletion replacement or transposition of input symbols, just like the unpractical global error recovery. However, to avoid the possibility of an infinite loop and to make it practical, only the prefix of the yet unrecognized part of the input is modified.

Example 11. *Consider parsing the erroneous expression* $\mathrm{num} + (\mathrm{num} + + \mathrm{num})$ *using the rules of the* LLR*-system of Example 6. Without error recovery, the parsing ends as*

$$\begin{aligned}
[\![\,\mathrm{num} + (\mathrm{num} + + \mathrm{num})\,]\!] &\Longrightarrow^* [\![\,E + (E + + \mathrm{num})]\!]\\
&\Longrightarrow [\![\,E + (E + +F)]\!] \Longrightarrow [\![\,E + (E + +T)]\!] \Longrightarrow [\![\,E + (E + +E)]\!]
\end{aligned}$$

as rule $E + T \longrightarrow E$ cannot be applied because of the surplus $+$. Be aware, however, that if the LLR*-system is used, the parsing can continue beyond the position of a syntax error: in this case, the third* num *is parsed even though it is encountered after the surplus $+$.*

To avoid reaching the sentential form $[\![\,E + (E + +E)]\!]$, one of the various error-recovery rules must be added to the LLR*-system. The list includes, among others, rules like $+ + \longrightarrow +$ (deletion) or $+ + \longrightarrow + \mathrm{num} +$ (insertion). It is upon a parser designer to choose one of these rules. If the former error recovery rule is added, the parsing starts as*

$$[\![\,\mathrm{num} + (\mathrm{num} + + \mathrm{num})\,]\!] \Longrightarrow^* [\![\,E + (E + + \mathrm{num})]\!] \Longrightarrow [\![\,E + (E + \mathrm{num})]\!],$$

and if the latter one is added, it starts as

$$[\![\,\mathrm{num} + (\mathrm{num} + + \mathrm{num})\,]\!] \Longrightarrow^* [\![\,E + (E + + \mathrm{num})]\!] \Longrightarrow [\![\,E + (E + \mathrm{num} + \mathrm{num})]\!].$$

In either case, the parsing successfully reaches the final sentential form $[\![\,E\,]\!]$.

It must be stressed that, without the error recovery, the parser still works correctly, as it still detects an error by not terminating with $[\![\,E\,]\!]$. Even if expressions are parsed within the bidirectional parser, the error is still detected as no rule $\overline{E}Ex \longrightarrow x$, where $x \neq +$, can be applied, and thus the sentential form cannot be reduced to $[\![\,S\,]\!]$.

The LLR-system has an ability to parse parts of the program taken out of their context. This has been used in parsing fragments of the Pascal and C/C++ code by WEB and

CWEB [13,14]. The same ability, as shown in the example above, allows the LLR-system to parse parts of its input even though a syntax error has been detected and left uncorrected. If language constructs are introduced by keywords assumed to be misused rarely or if it contains any other patterns that identify the start of these constructs [18,47], the parser can analyze major parts of the program, e.g., subroutines, declarations or statements, even if the text around them is incorrect to the point that the parser cannot recover from parsing it.

6. Evaluation

Programming languages are usually parsed with algorithms based on context-free grammars. However, if parsing is based on context-sensitive grammar or even type-0 grammar, the question arises about the price, which is paid when an algorithm that permits a much more general description of a language syntax is used. The comparison can, of course, only be made using languages that can be described by context-free grammars. Nevertheless, the results obtained using these languages serve as a good indicator of how efficient the parsing algorithm based on the LLR-system is compared to some more traditional parsing algorithm.

To evaluate parsing based on the LLR-system, a prototype implementation of the parser generator supporting all three algorithms has been made. Together with all the test data, it is available at https://github.com/slivnik/LLR-systems (accessed on 12 March 2023, commit ca5d3f5). All the results reported in this section were obtained using an Intel(R) Core(TM) i7-7700HQ processor running at 2.80 GHz (max 3.80 GHz).

Consider parsing arithmetic expressions described by the context-free grammar defined in Example 6 first. The running times of parsing a random arithmetic expression consisting of one million and one symbols (all nums are single digits and all ids are single letters) are shown in Table 2 and compared to the running time of the `bison`-generated parser. The first line in Table 2 refers to the LLR-system obtained by transforming the grammar from Example 6 to LL grammar first and then to an LLR-system. The second line refers to the LLR-system obtained by transforming the grammar from Example 6 directly to the LLR-system. The third line refers to the LLR-system written by hand in Example 6.

As expected, the LLR-system written from scratch yields better results than the two obtained by transforming the grammar using the transformations described in Section 4. It can be observed that the parser based on the LLR-system shown in Example 6 is 18% slower than the `bison`-generated parser. If both scanning and parsing are considered, as shown in Table 3, the LLR-system parser is only 6% slower. However, it is worth examining the absolute times too: the actual difference between the `bison`-generated parser and the LLR-system is approx. 2 ms of the CPU time for a 1 Mb source file.

Table 2. The running times of parsing an arithmetic expression consisting of a million symbols using the full backjumping, limited backjumping and DFA-based algorithm for the different LLR-systems derived from the context-free grammar in Example 6.

	Full Backjumping		Limited Backjumping		dfa-Based	
	Factor	**Time in Secs.**	**Factor**	**Time in Secs.**	**Factor**	**Time in Secs.**
LR $\Rightarrow$ LL $\Rightarrow$ LLR-system	67.86	0.769031 s	3.28	0.037158 s	2.49	0.028260 s
LR $\Rightarrow$ LLR-system	50.07	0.567409 s	3.78	0.042878 s	3.31	0.037537 s
hand-coded LLR-system	19.49	0.220924 s	2.45	0.027768 s	**1.18**	0.013390 s

`bison`: 0.011333 s.

Table 3. The running times of scanning and parsing an arithmetic expression consisting of a million symbols using the full backjumping, limited backjumping, and DFA-based algorithm for the different LLR-systems derived from the context-free grammar in Example 6.

	Full Backjumping		**Limited Backjumping**		**dfa-Based**	
	Factor	**Time in Secs.**	**Factor**	**Time in Secs.**	**Factor**	**Time in Secs.**
LR ⇒ LL ⇒ LLR-system	30.84	0.781970 s	1.97	0.049952 s	1.62	0.041080 s
LR ⇒ LLR-system	22.88	0.580191 s	2.19	0.055662 s	1.98	0.050320 s
hand-coded LLR-system	9.25	0.234479 s	1.61	0.040776 s	**1.06**	0.026818 s

`flex+bison`: 0.025359 s.

Parsing arithmetic expressions consisting of 10^6 symbols is a rather synthetic test. Hence, parsing Pascal, a real programming language, is considered next. Pascal has been chosen for two reasons that both make measuring running time simpler. First, unlike C, it can be parsed without a symbol table and thus the running time is easier to measure. Second, compared to Java, source files in Pascal are usually much bigger. To avoid generating random programs, source files of Knuth's `TANGLE`, `WEAVE`, TEX and METAFONT (all available on CTAN servers) were used as test inputs. There are two reasons for this choice. First, these are not some synthetic programs but publicly available ones of considerable size that will remain available for a very long time. Second, Pascal can be parsed without a symbol table which makes measurements much simpler (the only other part necessary is lexer).

The running times of two LLR-systems were compared against a `bison`-generated parser. The first one is based on the LL grammar for Pascal, but with a few LL conflicts resolved as described in Section 4. A small subset of its 278 rules is shown in Figure 3 (numbers on the right side of rules are substituted by symbols found at the specified positions on the left side). Rules expanding the symbols `program`, `label_part`, `label_definitions` and `label_definitions_rest` illustrate how rules based on an LL(1) grammar look like. However, having symbol `statement` on the top of the stack and `IDENTIFIER` in the lookahead buffer, the LL(1) parser cannot know whether it should start a parsing assignment statement or a procedure statement. As it can afford to check two input symbols instead of just one (without a hack in the parser's code), the decision can be made.

```
program (PROGRAM) -->
tDOT compound_statement subprogram_part variable_part type_part constant_part label_part program_header 2 ;

label_part (LABEL)                                    --> tSEMIC label_definitions tLABEL 2 ;
label_part (PROCEDURE|FUNCTION|VAR|CONST|TYPE|BEGIN) --> 2 ;

label_definitions (INTEGERCONST) --> label_definitions_rest tINTEGERCONST 2 ;
label_definitions_rest (COMMA)   --> label_definitions_rest tINTEGERCONST tCOMMA 2 ;
label_definitions_rest (SEMIC)   --> 2 ;

statement (IDENTIFIER) (ASSIGN|DOT|LBRACKET|PTR) --> assignment_statement 2 3 ;
statement (IDENTIFIER)                           --> procedure_statement 2 ;
```

Figure 3. A subset of rules of the LL-grammar-based LLR-system for parsing Pascal.

The other LLR-system has been made from scratch and is found to be more efficient (the same pattern as with the arithmetic expressions). A small subset of its 209 rules is shown in Figure 4. A totally different approach is used here. For instance, the third rule does not depend on the context: wherever `LABEL` is found, it triggers the parsing of label declarations. Moreover, if it is reduced to `label_declarations` right after the program header, it is merged together with it; otherwise, the program header changes to the same symbol, namely `program_upto_labels` without any label declarations. Hence, this LLR-system resembles the bottom-up shift-reduce parsing, but in a much more compact way, and a programmer can have a better understanding of its reductions than if he or she would inspect the reductions of an LR parser.

```
program_header label_declarations (CONST|TYPE|VAR|PROCEDURE|FUNCTION|BEGIN) --> program_upto_labels(1,2) 3;
program_header (CONST|TYPE|VAR|PROCEDURE|FUNCTION|BEGIN) --> program_upto_labels(1) 2 ;

LABEL label_names SEMIC        --> label_declarations(1,2,3) ;

LABEL INTEGERCONST             --> 1 label_names(2) ;
label_names COMMA INTEGERCONST --> label_names(1,2,3) ;
```

Figure 4. Parsing Pascal: a subset of rules of the LLR-system not resembling a grammar.

Tables 4 and 5 contain running times of parsing all four programs with all three algorithms. The running times are again compared to the bison-generated parser: the parsers based on the LLR-system that was made specifically for LLR-parsing are less than 60% slower if the CPU time of the parsing is compared and less than 20% percent if both scanning and parsing are considered.

Table 4. Parsing of TANGLE, WEAVE, TEX and METAFONT using two different LLR-systems; running times of parsing in seconds and comparison to the bison-generated parser.

	Full Backjumping		Limited Backjumping		dfa-Based	
	Factor	Time in Secs.	Factor	Time in Secs.	Factor	Time in Secs.
TANGLE						
LLR-system 1	16.77	0.004964 s	5.57	0.001648 s	2.45	0.000725 s
LLR-system 2	11.62	0.003440 s	3.28	0.000976 s	**1.47**	0.000436 s
WEAVE						
LLR-system 1	16.96	0.010498 s	5.67	0.000619 s	2.45	0.001518 s
LLR-system 2	12.49	0.007733 s	3.18	0.001936 s	**1.51**	0.000933 s
TEX						
LLR-system 1	16.13	0.041531 s	5.39	0.013884 s	2.41	0.006189 s
LLR-system 2	11.01	0.028322 s	3.04	0.007831 s	**1.44**	0.003696 s
METAFONT						
LLR-system 1	17.33	0.043925 s	5.82	0.014754 s	2.51	0.006358 s
LLR-system 2	11.87	0.030086 s	3.23	0.008187 s	**1.56**	0.003959 s

bison: 0.000296 s (TANGLE), 0.000619 s (WEAVE), 0.002574 s (TEX), 0.002534 s (METAFONT).

Table 5. Parsing of TANGLE, WEAVE, TEX and METAFONT using two different LLR-systems: running times of parsing and scanning in seconds and comparison to the bison-generated parser.

	Full Backjumping		Limited Backjumping		dfa-Based	
	Factor	Time in Secs.	Factor	Time in Secs.	Factor	Time in Secs.
TANGLE						
LLR-system 1	5.89	0.005610 s	2.41	0.002296 s	1.45	0.001376 s
LLR-system 2	4.30	0.004092 s	1.71	0.001626 s	**1.14**	0.001087 s
WEAVE						
LLR-system 1	6.20	0.011772 s	2.52	0.004783 s	1.47	0.002796 s
LLR-system 2	4.74	0.009011 s	1.69	0.003210 s	**1.16**	0.002203 s
TEX						
LLR-system 1	6.12	0.046536 s	2.48	0.018866 s	1.47	0.011180 s
LLR-system 2	4.38	0.033338 s	1.69	0.012818 s	**1.14**	0.008701 s
METAFONT						
LLR-system 1	5.92	0.049735 s	2.45	0.020571 s	1.45	0.012204 s
LLR-system 2	4.28	0.035925 s	1.68	0.014003 s	**1.16**	0.009786 s

flex+bison: 0.000952 s (TANGLE), 0.001898 s (WEAVE), 0.007603 s (TEX), 0.008400 s (METAFONT).

The difference in CPU time between parsing TEX and METAFONT using the bison-generated parser and the LLR-based parser is less than 2 ms if the more efficient LLR-system is used and less than 4 ms if the LL-based LLR-system is used. As TEX and METAFONT sources contain 19,203 and 18,769 lines of a pure Pascal code (excluding comments and formatted using `ptop`), extrapolating this to one million lines of code yields a difference of less than 0.1 s and approx. 0.2 s, respectively. If this was a real code with a proper amount of comments, the difference would be even smaller, as comments never reach the parser. Putting this into perspective, it takes the Free Pascal Compiler (version 3.2) 0.313 s to compile TEX (with a few lines of code commented out for the sake of incompatibility). Hence, if the `bison`-generated parser or LLR-based parser had been used, the difference would amount to 0.3% or 1.1% of the compilation time.

As proved in Appendix B, the LLR-systems are equivalent to Turing machines. Hence, comparing different LLR-systems and their complexity in general is a very difficult task. More insight, at least for the practical purposes, can be obtained by inspecting individual reductions of different LLR-systems for arithmetic reductions or for parsing Pascal (see https://github.com/slivnik/LLR-systems, accessed on 12 March 2023, commit ca5d3f5). For instance, in most cases, the introduction of a complementary symbol $\overline{a}$ for each terminal symbol $a \in T$ and a reduction $\overline{a}a \longrightarrow \varepsilon$ (as it is prescribed by the transformation of an LL grammar into an LLR-system) can be avoided. As with context-free grammars, it takes a bit of practice to gain the expertise needed for writing efficient parsers.

7. Conclusions

A new formalism, called the LLR-system, has been defined. As it is meant to be used for parsing programming and domain-specific languages, two sections, namely Sections 4 and 5, have been devoted to the definitions of initial patterns that can be used and followed when building parsers based on LLR-systems. It is hoped, at the same time, that the explanation of these patterns provides the insight into the structure of the LLR-system and into the way of thinking needed to build an efficient LLR-system. By providing these patterns and explanation, the LLR-system is perhaps given a better chance to become used in practice.

As shown in the evaluation (Section 6), parsers based on LLR-systems are slightly slower than classical context-free parsers. However, if their absolute running times are taken into consideration, the difference is negligible when compared to the difference in running times of code written in different programming languages. After all, it has become perfectly acceptable that code written in one of the most widely used programming languages nowadays is interpreted 70 times slower than when the equivalent compiled code is run when written in another language [63]. Hence, only a few people would notice or care if an interpreter or compiler used a parser that runs a 1–5 milliseconds longer even when large programs, e.g., the entire source code of TEX or METAFONT, are parsed.

Thus, if a parser can be made using an existing algorithm without too much additional code needed to support the parsing of the most complex language constructs, it should be made this way. Otherwise, assuming that someone is skilled enough and willing to implement a hand-coded recursive-descent parser, he or she can implement it as an LLR-system just as well. The advantage of implementing it as an LLR-system is that many add-ons or hacks can be implemented "within-a-system" and can be, if needed later, modified more easily. This is a significant step forward if compared with Markov normal algorithms which, as already mentioned in Section 3, "*did not meet general agreement*", because it is hard to write a set of adequate rules representing a parser [53].

Finally, it has been demonstrated how the LLR-system can be used for implementing the existing context-free parsing algorithms with error recovery, which is always language dependent, being part of the formal parser specification. Realizing the ability of the LLR-system to parse languages with considerably more complex syntax structures than are found in most existing programming and domain-specific languages, it would be interesting to observe how a generation of an LLR-system-based parser could be added to or integrated into some existing parser generator to observe how well the method works once syntax-directed definitions [18] are supported. One such candidate is LISA, a compiler–compiler capable of generating parsers supporting not only S- or L-attributed syntax-directed definitions, but attribute grammars as well [64].

Author Contributions: Conceptualization, B.S. and M.M.; Software, B.S. and M.M.; Validation, B.S. and M.M.; Formal analysis, B.S. and M.M.; Investigation, B.S. and M.M.; Writing—original draft, B.S. and M.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Slovenian Research Agency (Research Core Funding No. P2-0041).

Data Availability Statement: https://github.com/slivnik/LLR-systems (accessed on 12 March 2023, commit ca5d3f5).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. List of Abbreviations

Throughout the paper, abbreviations are used for various parsing algorithms. Table A1 contains a brief explanation of abbreviations. An interested reader is referred to the reference works for further information about each method. The formulation of LL, SLL, LR, and LALR parsers used in Section 4, i.e., as rewriting systems using shift and produce rules for LL and SLL parsing and shift and reduce rules for LR and LALR parsing, is explained in detail in [15,16].

Table A1. List of abbreviations.

CYK	Cook-Younger-Kasami (tabular) parsing algorithm
LL	(Top-down) shift-produce parsing method which scans the input from left to right and produces the leftmost derivation [15,16]
LL(k)	(The canonical) LL(k) parsing, uses the lookahead of k symbols [15,16]
SLL(k)	The strong LL(k) parsing, a simplification of the LL(k) parsing [15,16]
GLL(k)	The generalized LL(k) parsing, works with ambiguous grammars
ALL($*$)	An LL parsing which can extend the decision automaton in runtime [4,5]
LR	(Bottom-up) shift-reduce parsing method which scans the input from left to right and produces the rightmost derivation in reverse order [15,16]
LR(k)	(The canonical) LR(k) parsing, uses the lookahead of k symbols [15,16]
LALR(k)	The lookahead LL(k) parser, a simplification of the LR(k) parser [15,16]
GLR(k)	The generalized LL(k) parsing, works with ambiguous grammars
LLLR	A bidirectional parsing method, a combination of LL and LR parsing, produces the leftmost derivation [11]
PEG	Parsing Expression Grammars [46]

Appendix B. Theorem

The following theorem states that the LLR-system is Turing-complete. In other words, the restrictions imposed on the form of reductions and how reductions are used do not degrade its computational power. However, the theorem and its proof do not bring anything necessary for the understanding of the other parts of the paper and are included for the sake of completeness only.

Theorem A1. *L is accepted by a deterministic Turing machine if and only if L is accepted by an* LLR-*system.*

Proof. ($\Longrightarrow$) Consider a deterministic Turing machine $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F\rangle$ (as defined in [19]) and assume, without a loss of generality, that $\delta(q_f, X)$ is undefined for all $q_f \in F$ and $X \in \Gamma$. Define an LLR-system $\mathcal{R} = \langle V, \Sigma, R, S, [\![,]\!]\rangle$, where $V = Q \cup \Gamma \cup \{S, [\![,]\!], \bullet\}$, $\{S, [\![,]\!], \bullet\} \cap (Q \cup \Gamma\} = \emptyset$, which simulates the computation of $\mathcal{M}$ using a state to mark the position of $\mathcal{M}$'s tape head: the symbol "pointed to" by the tape head is the first symbol right of the state. To achieve this, R must include the following reductions:

1. *Introducing the initial state:*

$$\forall X \in V \setminus \{\bullet\}\colon \big(\,[\![X \longrightarrow [\![\bullet\, q_0 X\,\big) \in R$$

Each of these reductions starts the simulation of machine $\mathcal{M}$ in the initial state and with the tape head pointing to the leftmost symbol of the input string. Only one of these reductions will be used during the entire reduction process and only once: the sole purpose of symbol $\bullet$ is to ensure that $\mathcal{M}$ is not started multiple times.

2. *Ensuring the infinite tape:*

$$\forall q \in Q \setminus F\colon \big(\,q]\!] \longrightarrow qB]\!]\,\big) \in R$$

Whenever the state reaches the right marker, the tape head would have reached a new blank tape cell and therefore the sentential form is extended by another B.

3. *Simulating $\mathcal{M}$:*

$$\begin{aligned}\delta(q, X) = \langle q', Y, \textsc{r}\rangle &\implies \forall Z \in V \setminus \{\bullet\}\colon \big(\,qXZ \longrightarrow Yq'Z\,\big) \in R\\ \delta(q, X) = \langle q', Y, \textsc{l}\rangle &\implies \forall Z \in V \setminus \{\bullet\}\colon \big(\,ZqX \longrightarrow q'ZY\,\big) \in R\end{aligned}$$

Non-final states perform transitions as specified by $\mathcal{M}$ are possibly interrupted by extending the sentential form by a new B whenever needed.

4. *Cleaning up and accepting:*

$$\begin{aligned}&\forall q_F \in F, X \in V \setminus \{[\![,]\!], \bullet\}\colon \big(\,Xq_F \longrightarrow q_F\,\big) \in R\\ &\forall q_F \in F, X \in V \setminus \{[\![,]\!], \bullet\}\colon \big(\,q_F X \longrightarrow q_F\,\big) \in R\\ &\forall q_F \in F\colon \big(\,[\![\bullet\, q_F]\!] \longrightarrow [\![S]\!]\,\big) \in R\end{aligned}$$

The final state first deletes all symbols of the sentential form and finally announces acceptance by mutating into S.

It is shown by induction on the length of the $\mathcal{M}$'s computation that $\mathcal{R}$ actually simulates $\mathcal{M}$ and accepts input if and only if $\mathcal{M}$ accepts it. As this part of the proof is a matter of routine (and is basically the same as the corresponding part of the proof of Theorem 3 in [8]), it is left as an exercise.

($\Longleftarrow$) The LLR-system can be implemented on a digital computer (up to the limit of the available memory needed to store the sentential form). Therefore, the deterministic Turing machine can simulate it (with all the memory needed on its infinite tape). $\square$

References

1. Gosling, J.; Joy, B.; Steele, G. *The Java Language Specification*, 1st ed.; The Java Series; Addison-Wesley: Reading, MA, USA, 1996.
2. Gosling, J.; Joy, B.; Steele, G.; Bracha, G. *The Java Language Specification*, 3rd ed.; The Java Series; Addison-Wesley: Reading, MA, USA, 2005.
3. Medeiros, C.M.; Musicante, M.A.; Costa, U.S. LL-based query answering over RDF databases. *J. Comput. Lang.* **2019**, *51*, 75–87. [CrossRef]
4. Parr, T.; Harwell, S.; Fischer, K. Adaptive LL(*) parsing: The power of dynamic analysis. *ACM SIGPLAN Not.* **2014**, *49*, 579–598. [CrossRef]
5. Parr, T. *The Definitive ANTLR 4 Reference*; The Pragmatic Bookshelf: Dallas, TX, USA, 2014.
6. Mendelson, E. *Introduction to Mathematical Logic*, 6th ed.; CRC Press: Boca Raton, FL, USA, 2015.
7. Robič, B. *The Foundations of Computability Theory*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2020.
8. Slivnik, B. Context-sensitive parsing for programming languages. *J. Comput. Lang.* **2022**, *73*, 101172. [CrossRef]
9. Mernik, M.; Heering, J.; Sloane, A. When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* **2005**, *37*, 316–344. [CrossRef]
10. Kosar, T.; Bohra, S.; Mernik, M. Domain-Specific Languages: A Systematic Mapping Study. *Inf. Softw. Technol.* **2016**, *71*, 77–91. [CrossRef]
11. Slivnik, B. LLLR Parsing: A Combination of LL and LR parsing. In Proceedings of the 5th Symposium on Languages, Applications and Technologies (SLATE'16), Maribor, Slovenia, 20–21 June 2016; pp. 5:1–5:13.

12. Slivnik, B. On different LL and LR parsers used in LLLR parsing. *Comput. Lang. Syst. Struct.* **2017**, *50*, 108–126. [CrossRef]
13. Knuth, D.E. The WEAVE Processor (Version 4.5). Comprehensive TEX Archive Network (CTAN). 2021. Available online: https://ctan.org/pkg/web (accessed on 10 March 2022).
14. Levy, S.; Knuth, D.E. The CWEAVE Processor (Version 4.7). Comprehensive TEX Archive Network (CTAN). 2022. Available online: https://ctan.org/pkg/cweb (accessed on 10 March 2022).
15. Sippu, S.; Soisalon-Soininen, E. *Parsing Theory, Volume I: Languages and Parsing*; Springer: Berlin/Heidelberg, Germany, 1988; Volume 15.
16. Sippu, S.; Soisalon-Soininen, E. *Parsing Theory, Volume II: LR(k) and LL(k) Parsing*; Springer: Berlin/Heidelberg, Germany, 1990; Volume 20.
17. Grune, D.; Jacobs, C.J. *Parsing Techniques, A Practical Guide*, 2nd ed.; Monographs in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008.
18. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compiler—Principles, Techniques, and Tools*, 2nd ed.; Pearson Education: Boston, MA, USA, 2007.
19. Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*; Addison-Wesley: Reading, MA, USA, 1979.
20. Appel, A.W.; Palsberg, J. *Modern Compiler Implementation in Java*, 2nd ed.; Cambridge University Press: Cambridge, UK, 2004.
21. Gazzillo, P.; Grimm, R. SuperC: Parsing All of C by Taming the Preprocessor. *ACM SIGPLAN Not.* **2012**, *47*, 323–334. [CrossRef]
22. Aycock, J.; Horspool, R.N. Schrödinger's token. *Softw. Pract. Exp.* **2001**, *31*, 803–814. [CrossRef]
23. Van Wyk, E.R.; Schwerdfeger, A.C. Context-Aware Scanning for Parsing Extensible Languages. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), Salzburg, Austria, 1–3 October 2007; pp. 63–72.
24. Leber, Ž.; Črepinšek, M.; Mernik, M.; Kosar, T. RNGSGLR: Generalization of the Context-Aware Scanning Architecture for All Character-Level Context-Free Languages. *Mathematics* **2022**, *10*, 2436. [CrossRef]
25. Régis-Gianas, Y.; Jeannerod, N.; Treinen, R. Morbig: A Static parser for POSIX shell. *J. Comput. Lang.* **2020**, *57*, 100944. [CrossRef]
26. Levine, J.; Brown, D.; Mason, T. *lex & yacc*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 1992.
27. Kurš, J.; Vraný, J.; Ghafari, M.; Lungu, M.; Nierstrasz, O. Efficient parsing with parser combinators. *Sci. Comput. Program.* **2018**, *161*, 57–88. [CrossRef]
28. Willis, J.; Wu, N.; Pickering, M. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* **2020**, *4*, 120. [CrossRef]
29. Thurston, A.D.; Cordy, J.R. A Backtracking LR Algorithm for Parsing Ambiguous Context-Dependent Languages. In Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'06), Toronto, ON, Canada, 16–19 October 2006; pp. 4–18.
30. Laurent, N.; Mens, K. Taming Context-Sensitive Languages with Principled Stateful Parsing. In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16), Amsterdam, The Netherlands, 31 October–1 November 2016; pp. 15–27.
31. Jim, T.; Mandelbaum, Y. Efficient Earley Parsing with Regular Right-hand Sides. *Electron. Notes Theor. Comput. Sci.* **2010**, *253*, 135–148. [CrossRef]
32. Jim, T.; Mandelbaum, Y.; Walker, D. Semantics and Algorithms for Data-Dependent Grammars. *ACM SIGPLAN Not.* **2010**, *45*, 417–430. [CrossRef]
33. Afroozeh, A.; Izmaylova, A. One Parser to Rule Them All. In Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!15), Pittsburg, PA, USA, 20–22 October 2015; pp. 151–170.
34. Loff, B.; Moreira, N.; Reis, R. The computational power of parsing expression grammars. *J. Comput. Syst. Sci.* **2020**, *111*, 1–21. [CrossRef]
35. Grimm, R. Better Extensibility through Modular Syntax. *ACM SIGPLAN Not.* **2006**, *41*, 38–51. [CrossRef]
36. Kuramitsu, K. Nez: Practical open grammar language. In Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!16), Amsterdam, The Netherlands, 2–4 November 2016; pp. 29–42.
37. Matsumura, T.; Kuramitsu, K. A Declarative Extension of Parsing Expression Grammars for Recognizing Most Programming Languages. *J. Inf. Process.* **2016**, *24*, 256–264. [CrossRef]
38. Kuramitsu, K. A Symbol-Based Extension of Parsing Expression Grammars and Context-Sensitive Packrat Parsing. In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17), Vancouver, BC, Canada, 23–24 October 2017; pp. 26–37.
39. Hutton, G.; Meijer, E. Functional Pearl: Monadic parsing in Haskell. *J. Funct. Program.* **1998**, *8*, 437–444. [CrossRef]
40. Woods, W.A. Context-Sensitive Parsing. *Commun. ACM* **1970**, *13*, 437–445. [CrossRef]
41. Hadlock, F. A Context Sensitive Tabular Parsing Algorithm. In Proceedings of the 28th Annual Southeast Regional Conference, Greenville, SC, USA, 26–29 April 1990; pp. 111–117.
42. Schuler, P.F. Weakly context-sensitive languages as model for programming languages. *Acta Inform.* **1974**, *3*, 155–170. [CrossRef]
43. Ikeda, M.; Tanaka, E. A recognizing algorithm for a loop free context-sensitive language based on the dynamic programming method. *Syst. Comput. Jpn.* **1987**, *18*, 1–13. [CrossRef]
44. Knuth, D.E. On the Translation of Languages from Left to Right. *Inf. Control* **1965**, *8*, 607–639. [CrossRef]

45. Lewis II, P.M.; Stearns, R.E. Syntax-Directed Transduction. *J. ACM* **1968**, *15*, 465–488. [CrossRef]
46. Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages POPL'04, Venice, Italy, 14–16 January 2004; pp. 111–122.
47. Wirth, N. *Algorithms + Datat Structures = Programs*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1976.
48. Dejanović, D. Parglare: A LR/GLR parser for Python. *Sci. Comput. Program.* **2022**, *214*, 102734. [CrossRef]
49. Queiroz de Medeiros, S.; Mascarenhas, F. Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine. *J. Vis. Lang. Comput.* **2018**, *49*, 17–28. [CrossRef]
50. Queiroz de Medeiros, S.; de Azevedo Alvez Junior, G.; Mascarenhas, F. Automatic syntax error reporting and recovery in parsing expression grammars. *Sci. Comput. Program.* **2020**, *187*, 102373. [CrossRef]
51. De Jonge, M.; Kats, L.C.L.; Visser, E.; Söderberg, E. Natural and Flexible Error Recovery for Generated Modular Language Environments. *ACM Trans. Program. Lang. Syst.* **2012**, *34*, 15. [CrossRef]
52. Swierstra, S.D. Combinator Parsers: From Toys to Tools. *Electron. Notes Theor. Comput. Sci.* **2001**, *41*, 38–59. [CrossRef]
53. Leoni, G.; Sprugoli, R. Some relations between Markov algorithms and formal languages. *Calcolo* **1977**, *14*, 261–284. [CrossRef]
54. Aguzzi, G. The Theory of Invertible Algorithms. *R.A.I.R.O. Inform. Théor./Theor. Inform.* **1981**, *15*, 253–279.
55. Kushner, B.A. Markov's constructive analysis; A participant's view. *Theor. Comput. Sci.* **1999**, *219*, 267–285. [CrossRef]
56. Katzenelson, J. The Markov algorithm as a language parser—Linear bounds. *J. Comput. Syst. Sci.* **1972**, *6*, 465–478. [CrossRef]
57. Friedl, J.E.F. *Mastering Regular Expressions*, 3rd ed.; O'Reilly Media: Sebastopol, CA, USA, 2006.
58. Okasaki, C. Purely functional random-access lists. In Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95), La Jolla, CA, USA, 25–28 June 1995; pp. 86–95.
59. Aho, A.V.; Ullman, J.D. *The Theory of Parsing, Translation and Compiling*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1973; Volume 2.
60. Demers, A.J. Generalized Left Corner Parsing. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages POPL'77, Los Angeles, CA, USA, 20–21 January 1977; pp. 170–182.
61. Horspool, R.N. Recursive Ascent-Descent Parsing. *Comput. Lang.* **1993**, *18*, 1–16. [CrossRef]
62. Nederhof, M.J. Generalized Left Corner Parsing. In Proceedings of the Sixth Conference on European Chapter of the Association for Computational Linguistics EACL'93, Stroudsburg, PA, USA, 21–23 April 1993; pp. 305–314.
63. Pereira, R.; Couto, M.; Ribeiro, F.; Rua, R.; Cunha, J.; Fernandez, J.P.; Saraiva, J. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE'17), Vancouver, BC, Canada, 23–24 October 2017; pp. 256–267.
64. Mernik, M.; Žumer, V.; Lenič, M.; Avdičaušević, E. Implementation of Multiple Attribute Grammar Inheritance in the Tool LISA. *SIGPLAN Not.* **1999**, *34*, 68–75. [CrossRef]