BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

Parsing Based on Several Grammars

Daniel Ulvr*

Abstract
Compilers might parse different parts of a programming language most efficiently using different parsing methods. This paper introduces *CTCD grammar systems* – a new type of Cooperating Distributed (CD) grammar systems controlled by a *communication table* – and demonstrates their practical usage in such scenarios. A concrete eight-component CTCD grammar system is designed for a programming language inspired by Zig, and a parser is implemented for it that combines LL, operator-precedence, and SLR analysis within a single framework. The result is a working parser whose used analysis methods can be selected for individual components, making CTCD grammar systems a practical tool for multi-method parser construction.

*xulvrda00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Real-world programming languages include various syntactic constructs. These are naturally suited to different parsing methods. Arithmetic expressions, for instance, are most elegantly handled by LR or operator-precedence methods, while statement-level structures often fit perfectly into LL grammars. However, there is no unified approach to combining them into a single parser. Formal models that can express such cooperation of parsing methods may therefore bring practical value beyond their theoretical level. Grammar systems represent one such approach.

Cooperating Distributed (CD) grammar systems (see [1]) provide a formal model in which several grammars (*components*) share a common sentential form and take turns rewriting it. The central open question for their use in parsing is: how to specify the *cooperation protocol* – i.e., which component is active at each moment and for how long does it rewrite the sentential form. The cooperation protocol must also be *deterministic* for meaningful use. Without a deterministic protocol, CD grammar systems cannot be utilized within real-world parsers.

Existing related work from FIT BUT [2, 3, 4, 5] has explored various mechanisms, yet they share a common limitation: a single system component can be activated by at most one communication symbol, thereby restricting the protocol's expressiveness.

The *Communication Table-controlled CD* (CTCD) grammar systems are introduced as a new approach to solve the described challenges. Their *communication table* maps pairs of *communication nonterminals* and *communication terminals* to grammar components and can be constructed automatically from the components themselves via a simple algorithm (see **Algorithm 1**). The active component changes only when it can no longer rewrite the current sentential form, following the *terminating derivation mode*.

Furthermore, as a proof of concept, a CTCD grammar system with eight components is designed for a Zig-inspired language, featuring expressions with 16 operators, user-defined structs, `switch` expressions, and more.

Based on this system, a C++ parser is implemented to analyze the syntactic correctness of the designed language, along with lexical analysis and AST construction. The parser combines LL, operator-precedence, and SLR parsing. In addition, the user can select the parsing method per component at the start of the program.

2. CTCD Grammar Systems

A CTCD grammar system of degree n is a $(n+4)$-tuple: $\Gamma = (N, T, S, R, G_1, \ldots, G_n)$ where N, T, S are the usual nonterminal alphabet, terminal alphabet, and start symbol, each G_i is a finite set of context-free rewriting rules (a *component*), and $R \subseteq M \times C$ is the *communication table*.

The set M of *communication nonterminals* contains every nonterminal that appears on the left-hand side of

some rule in component G_i while also appearing on the right-hand side of a rule in some other component G_j ($i \neq j$). The set C of *communication terminals* contains every terminal that can appear as the leftmost terminal derived from any communication nonterminal (the *Predict* set). The *evaluation function* $\alpha(A, t) = G_i$ is defined whenever $t \in \mathit{Predict}(A \to x)$ and $A \to x \in G_i$.

There are two types of derivation steps. A *rewriting step* applies a rule of the currently active component to the leftmost nonterminal it can rewrite and is performed preferentially. A *control step* changes the active component based on the communication table. It is performed only when no rewriting step is possible, and $\alpha(A, t)$ is defined for the leftmost communication nonterminal A and the current communication terminal t. A *communication stack* records the chain of component changes so that control returns to the correct component after each sub-derivation (see **Figure 2**).

3. Application in Syntax Analysis

3.1 Designed Grammar System

The concrete CTCD grammar system designed for the Zig-inspired language has eight components (see **Figure 1**):

- G_1 – top-level structure (sequence of function and struct definitions)
- G_2 – function definitions, most statements (variable definition, assignment, `if-elif-else`, `while`, `for`, `return`), data types
- G_3 – comma-separated function parameters
- G_4 – function call command with arguments
- G_5 – `switch` expressions
- G_6 – user-defined struct definitions (attributes, default values, methods)
- G_7 – struct instantiation with attribute initialization
- G_8 – expressions with 16 operators (arithmetic, relational, logical, member access `.`)

Components G_1–G_7 are LL grammars, G_8 is an unambiguous CFG for expressions. G_1–G_7 can be analyzed by LL or SLR, G_8 by operator-precedence or SLR. The eight communication nonterminals in the communication table (see **Table 1**) include `<FUN_DEF>` and `<TYPE>`, both targeting G_2 but via different communication terminals – demonstrating that a single component can be activated by multiple communication nonterminals.

3.2 Parser Implementation

The parser is implemented in C++20 and is structured into nine modules (lexer, three analyzer classes, grammar system, AST, argument parser, utilities, and `main()` entry point). The abstract base class `SyntaxParser` exposes a virtual `parse()` method and a `communicate()` method that implements the communication stack. Concrete subclasses `LLParser`, `PrecedenceParser`, and `LRParser` provide the respective parsing methods (see **Figure 3**).

The runtime assignment of a parsing method to each component is handled by `ArgParser` and is stored in equivalent `Component` structs inside `GrammarSystem`. This design makes the three analyzers *mutually independent*: changing the method for one component does not affect the others.

The `LRParser` holds nine separate SLR tables (one per derivation sub-tree root, two of them for G_2), generated with an online SLR table tool and converted to C++ arrays by a Python script.

Lexical analysis was implemented for practical reasons, as it processes the input file and complements the parser.

In addition to the analysis part of the program, the AST representation of the input text structure is generated and can be exported in DOT format for visualization. An example of a generated AST is in **Figure 4**.

4. Conclusions

CTCD grammar systems aim to fill the gap between pure formal language theory and compiler construction.
Their communication table is formally defined, automatically constructible, and it groups multiple parsing methods into a single cooperating framework. The implemented C++ parser demonstrates this in practice. Function calls triggered by communication nonterminals perform a seamless change between individual parsers, which is entirely controlled by the communication table.

Future work could incorporate semantic analysis and code generation with the parser framework, thus creating a complete compiler for the designed language. Studying the formal generative power of CTCD systems in relation to the classical CD grammar system families might also be worthwhile.

Acknowledgements

I would like to thank my supervisor prof. RNDr. Alexandr Meduna, CSc., for his professional guidance and valuable advice throughout this work.

References

[1] Grzegorz Rozenberg and Arto Salomaa. *Handbook of Formal Languages: Volume 2: Linear Modeling: Background and Application*. Springer, Berlin, 1997.

[2] Matej Kunda. Systémy syntaktických analyzátorů. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2022.

[3] Adam Sedmík. Syntaktická analýza založená na gramatických systémech. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2022.

[4] Tomáš Repík. Systems of sequential grammars applied to parsing. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2014.

[5] Jakub Reš. Gramatické systémy aplikované v překladačích. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, Brno, 2021.