COMPUTER SCIENCE 2025. Vol. 35. Issue 2. Pp. 315–334.

MSC2020: 68N15

© *M. Joudakizadeh, A. P. Beltiukov*

PROGRAMMING IN GRAMMARS

This paper discusses an approach to programming based on the use of parameterized grammars. The concepts of these grammars are equipped with parameters that can also be objects of the grammars. Such grammars are quite a powerful tool; they are proposed to be used for formulating statements of problems of transformation of linguistic data. These grammars can be used directly for information processing, but this may not be effective. Therefore, a special class of such grammars that are effective in application is distinguished. A special system of unambiguous (functional) parameterized grammars is proposed, which can be used as an effective programming language for linguistic tasks. The ideas of deductive synthesis of programs within this system are described as deriving programs from problem statements in general parameterized grammars through logical inference, with the prospect of subsequent automation. The practical application of the proposed tool is demonstrated through examples of processing logical formulas and solving other problems. This work continues the ideas of Valentin Turchin in the field of the REFAL language.

Keywords: attribute grammars, parametric grammars, parameterized grammars, language programming, symbolic transformations, artificial intelligence, machine learning.

DOI: 10.35634/vm250210

Introduction

In the rapidly evolving landscape of artificial intelligence and computational linguistics, the ability to process, analyze, and transform linguistic data efficiently has become a cornerstone of technological advancement. From natural language understanding and machine translation to automated reasoning and symbolic computation, the demand for robust and flexible tools to handle language-related tasks is greater than ever. However, despite significant progress in the field, a fundamental challenge persists: the tension between the simplicity of traditional grammatical formalisms and their limited expressive power when applied to complex linguistic phenomena.

Traditional context-free grammars, particularly those expressed in Backus–Naur form (BNF), have long been celebrated for their elegance and intuitive clarity. These grammars provide a straightforward framework for describing the syntactic structure of languages, making them accessible and widely adopted in both theoretical and practical applications. Yet, their simplicity comes at a cost. While they excel at capturing basic language constructs, they fall short when it comes to modeling deeper semantic structures, handling context-sensitive transformations, or performing advanced linguistic operations. This limitation restricts their applicability to a narrow range of tasks, leaving more complex problems — such as semantic parsing, language translation, and automated reasoning — beyond their reach.

On the other hand, more powerful grammatical formalisms, such as attribute grammars, tree-adjoining grammars, and other extended frameworks, offer the theoretical capability to address these challenges. These approaches can encode complex syntactic and semantic relationships, enabling the representation of intricate language constructs and the execution of sophisticated transformations. However, their increased expressiveness often comes at the expense of practicality. The added complexity of these formalisms can lead to inefficiencies in implementation, difficulties in integration with modern software systems, and challenges in maintaining clarity and usability. As a result, while they hold great promise in theory, their real-world application is often hindered by these practical shortcomings.

This contradiction between simplicity and expressiveness lies at the heart of the problem we aim to address. Traditional grammars are easy to use but lack the power to handle complex tasks, while more advanced formalisms are powerful but often impractical for real-world applications. This tension is further compounded by the need for efficient processing in modern computational environments, where performance and scalability are critical. Directly applying parameterized grammars to linguistic data processing may yield expressive solutions, but their practical performance often falls short of the requirements for real-world applications. This highlights the need for a specialized subclass of grammars that can preserve the expressive power of advanced formalisms while enabling efficient computation, bridging the gap between theoretical capability and practical utility.

The motivation for our research stems from the desire to reconcile these opposing forces — simplicity and expressiveness, theoretical power and practical efficiency — by developing a unified programming paradigm that balances these competing demands. Our approach is inspired by the pioneering work of Valentin Turchin on the REFAL language, which demonstrated the potential of grammar-based programming for symbolic computation. REFAL introduced a revolutionary approach to programming by treating grammars as executable constructs, enabling powerful symbolic transformations and pattern matching. However, despite its theoretical significance, REFAL and similar languages have not achieved widespread adoption in practical programming. The reasons for this are multifaceted, including challenges in adapting REFAL to modern software platforms and inefficiencies in executing its programming constructs.

In light of these circumstances, we decided to return to the roots of formal language theory, specifically grammars in Backus–Naur form, while significantly expanding their syntax and semantics to address the limitations of traditional approaches. By doing so, we consciously sacrifice some of the power of REFAL's constructs, such as its mechanism of pattern matching through list traversal, in favor of a more structured and efficient formalism. Our approach transitions from classical context-free grammars to a more powerful framework of **parameterized grammars**, where grammatical concepts can be equipped with parameters that themselves serve as objects generated by the grammars. This allows us to develop a fundamentally new approach to programming linguistic tasks, where parameterized grammars, under certain constraints, naturally become executable programs.

A significant advantage of the proposed approach is its universality and flexibility in solving computational linguistic tasks. Parameterized grammars allow for the natural description of both syntactic and semantic aspects of linguistic constructions. Unlike traditional programming methods, which often require ad hoc solutions for different tasks, our approach provides a unified framework for describing languages, transforming linguistic structures, and performing semantic analysis. This flexibility is particularly valuable in applications such as language translation, where the ability to handle both syntax and semantics in a single framework can significantly simplify the development of language processors and translators.

At the core of our approach is the concept of **parameterized grammars**, which extend the classical formalism by incorporating parameters into grammatical constructs. These parameters serve not only to enrich the syntactic description but also to capture semantic nuances, enabling a unified framework for language description, transformation, and analysis. By allowing parameters to modify both syntactic structures and semantic interpretations, we facilitate the automatic, deductive synthesis of programs. This approach simplifies the development of language processors and compilers while ensuring that the resulting implementations are both correct and efficient. Furthermore, the ability to convert these grammar-based programs into conventional programming languages like JavaScript and Python broadens their practical applicability and enables integration with modern development ecosystems.

The potential impact of this research is far-reaching. By merging classical theories with con-

temporary computational techniques, our approach offers a unified and flexible framework for programming linguistic tasks. It promises significant advancements in the efficient processing, transformation, and semantic analysis of natural languages, paving the way for next-generation applications in computational linguistics and artificial intelligence. From natural language understanding and machine translation to automated reasoning and symbolic computation, the ability to handle complex linguistic tasks with both clarity and efficiency has the potential to revolutionize a wide range of fields.

In summary, our research is motivated by the need to overcome the trade-off between simplicity and expressiveness in grammatical formalisms, creating a formalism that is both theoretically powerful and practically efficient. By developing a unified programming paradigm that balances these competing demands, we aim to provide a robust and flexible tool for addressing the complex linguistic challenges of the modern era. This work not only builds on the rich legacy of formal language theory but also pushes the boundaries of what is possible in computational linguistics, offering new opportunities for innovation and discovery in the field.

§ 1. Background

The evolution of programming languages has been shaped by the continuous struggle to balance expressiveness and simplicity. On one hand, highly expressive formalisms facilitate complex data transformations but often sacrifice usability; on the other hand, simpler languages offer accessibility but limit the scope of solvable problems. This tension is particularly evident in the domain of linguistic data processing, where traditional approaches have repeatedly fallen short. The theoretical underpinnings of our work trace back to Chomsky's [1] foundational research on formal properties of grammars, which established the hierarchy of formal languages that continues to influence computational linguistics. Backus [2] further advanced the field by introducing a notation for describing language syntax that would later evolve into the Backus–Naur Form (BNF), a cornerstone of language specification. Context-free grammars, while elegant in their simplicity, exhibit fundamental limitations when applied to complex linguistic phenomena. As Knuth [3] demonstrated in his seminal work on the semantics of context-free languages, traditional grammatical formalisms struggle to capture the rich semantic structures inherent in both natural and programming languages. This limitation has spurred research into more powerful extensions of basic grammar models.

The concept of using grammars as programming tools has a rich history. Hehner and Silverberg [4] pioneered the exploration of grammar-directed language design, advocating for a methodological approach where programming languages emerge naturally from grammatical structures. Their work demonstrated how formal grammars could serve not merely as descriptive tools but as prescriptive frameworks for language implementation. Attribute grammars, extensively surveyed by Paakki [5], emerged as a powerful extension to context-free grammars. By augmenting grammatical rules with attributes and semantic equations, these formalisms provided a mechanism for expressing both syntactic structure and semantic interpretation within a unified framework. This advancement represented a significant step toward bridging the gap between grammar specification and practical language implementation. The SNOBOL family of languages, chronicled by Griswold [6], demonstrated the practical potential of pattern-matching approaches to programming. These languages introduced innovative constructs for text processing and manipulation, showing how grammatical concepts could be embedded directly into programming paradigms to facilitate linguistic operations.

A pivotal influence on our research has been the groundbreaking work of Valentin Turchin, particularly his development of the REFAL programming language [7] and the concept of supercompilation [8]. REFAL introduced a revolutionary approach to programming by treating grammars as executable constructs, enabling powerful symbolic transformations and pattern matching.

The language's emphasis on recursion and pattern-based substitution provided a natural framework for expressing complex transformations of symbolic data. Turchin's concept of a supercompiler [8] further extended these ideas by introducing techniques for program transformation and optimization. The supercompiler analyzes programs to identify opportunities for improvement, generating more efficient implementations while preserving semantic equivalence. This work, later advanced by Nemytykh [9] and adapted to modern languages by Klimov [10], demonstrated the potential for automated program synthesis and transformation based on grammatical principles. Romanenko's [11] exploration of inversion and metacomputation expanded upon Turchin's foundation, developing techniques for reasoning about and manipulating computational processes. These advancements in metacomputation — the ability of programs to operate on representations of themselves — established critical theoretical groundwork for our approach to program synthesis from grammatical specifications.

More recent work has continued to explore the intersection of grammars and programming. Besova et al. [12] demonstrated how grammatical principles could be applied to model transformations in software engineering, providing evidence for the broader applicability of grammar-based techniques beyond traditional language processing. Carette and Kiselyov [13] addressed the efficiency challenges in generic programming through multi-stage programming with functors and monads. Their work highlighted the importance of eliminating abstraction overhead, a concern that resonates with our focus on developing practical, efficient implementations of grammar-based programs. Despite these advances, significant challenges remain in reconciling the expressive power of advanced grammatical formalisms with the practical requirements of modern software development. Direct application of parameterized grammars to linguistic data processing, while theoretically appealing, often produces inefficient implementations unsuitable for real-world deployment. This tension between theoretical capability and practical utility underscores the need for our research.

Current approaches to programming linguistic tasks typically fall into one of two categories: they either utilize powerful but unwieldy grammatical formalisms that sacrifice efficiency and usability, or they employ ad hoc solutions that lack theoretical grounding and generality. This dichotomy highlights a critical gap in the field — the absence of a unified framework that balances expressiveness and practicality. Traditional programming languages, while efficient, often lack the natural facilities for expressing linguistic transformations. Conversely, specialized language processing tools may provide domain-specific capabilities but fail to integrate seamlessly with broader software ecosystems. This fragmentation of approaches hampers the development of comprehensive solutions to complex linguistic problems. Our research aims to bridge this gap by returning to the fundamental principles of formal language theory while incorporating the insights gained from decades of research in programming language design, attribute grammars, and program transformation. By developing a specialized class of parameterized grammars that maintain expressiveness while enabling efficient implementation, we seek to provide a practical framework for programming linguistic tasks that combines theoretical rigor with practical utility.

§ 2. Parameterized grammars

2.1. Metasyntax of parameterized grammars

By metasyntax, we mean the syntax for writing the grammars themselves. A parameterized grammar is defined by a set of rules described by the syntactic structure below. The description of the context-free metasyntax of rules (the concept <rule>) is given below in the form of Backus–Naur forms. To avoid confusion when reading, chains of terminal symbols are enclosed in apostrophes (' '). Non-terminal symbols are enclosed in angle brackets (< >).

The meta-rules for generating rules of a parameterized grammar are as follows:

```
<rule> ::= '<'<name> ' ' <expression>'>'
                                '='<expression> <descriptions> ';',
<descriptions> ::= | ',' <name> ':' <expression> <descriptions>.
```

Here, the symbol "::=" is the separator between the left and right parts of the metarule, and '=' is the separator between parts of the rule described by the grammar. In this form, it is necessary to distinguish between angle brackets without apostrophes, which limit metaconcepts, and angle brackets with apostrophes, which limit the concepts of the described grammar and their parameters. The metaconcept <name> is the name of the described concept, a non-empty string of letters. The metaconcept <expression> is an expression whose syntax is described below.

Thus, any rule of a parameterized grammar has the form:

$$< N\ t >= u, v_1 : w_1, \ldots, v_n : w_n;$$

where

- N is a non-terminal symbol (the name of the concept),
- expressions t and u are the parameter and the right part of this rule, respectively,
- v_i are variables (names, identifiers),
- w_i are expressions that specify the permissible values of the corresponding variables.

In the definitions of permissible values (types) of variables $v_i : w_i$, circular chains of references are prohibited to avoid logical difficulties. The value of n can be zero. In that case, a semicolon follows immediately after u.

The context-free metasyntax of expressions of parameterized grammars <expression> is written in Backus–Naur form as follows:

```
<expression> ::= | <term> <expression>;

<term> ::= <quote> <string> <quote>
| '(' <expression> ')'
| <name> ' '
| '<' <name> ' ' <expression> '>';
```

where

- <quote> is an apostrophe, the symbol ';
- <string> is a string (a non-empty chain of arbitrary elements from the considered alphabet of terminal symbols of the grammar, not containing apostrophes; we assume that certain classes of grammars are considered, and for each of these considered classes, a certain alphabet of terminal symbols is fixed),
- <name> are names (non-empty chains of letters).

Names in expressions after the sign '<' are concept names, while other names are local variables of the rules. We use letters from a pre-selected alphabet for names. In this work, we use lowercase English letters. Consecutive names are separated by spaces. Trailing spaces in names may be omitted when writing. For readability, we will insert additional spaces and line breaks, but not within names.

2.2. Semantics of parameterized grammars

The semantics of a parameterized grammar is defined as follows. The membership of a string d in the set defined by the expression $< N\ e >$ means that there exists a rule in the grammar of the form

$$< N\ t >= u, v_1 : w_1, \ldots, v_n : w_n,$$

for which values of the variables are selected in the form of chains of symbols such that:

- the string e belongs to the set generated by the expression t,
- the string d belongs to the set generated by the expression u,
- the chains v_i belong to the sets generated by the expressions w_i, respectively.

If no set of permissible values is specified for some variable, it is assumed that this variable takes values from some standard set chosen for the considered class of grammars. By default, this is any chain of terminal symbols. When generating chains from expressions, some representatives of the participating concepts with parameters are taken, and the implied operations of concatenating chains are performed. This definition is recursive. If there is no process of its application that terminates in a finite number of steps, then it should be considered that the given chain does not belong to the given set.

2.3. Example of a parameterized grammar generating some predicate logic formulas

Let us give an example of a grammar that generates a set of predicate logic formulas with permissible free variables from a given list.

For simplicity, we will limit ourselves to the following possible elements of formulas:

- one type of quantifier (the universal quantifier, denoted by the letter "A"),
- one type of binary connective (conjunction " $\&$ "),
- two predicates (a unary "P" and a binary "Q"),
- three subject variables ("x", "y", "z"),
- negation (denoted by the sign "$-$").

Subject constants are not used.

Next, the expression <formula w> means "a formula with free variables from the list w". The corresponding parameterized grammar looks as follows:

```
<formula w> = 'P(' <element w> ')';
<formula w> = 'Q(' <element w> ', ' <element w> ')';
<formula w> = '-' <formula w>;
<formula w> = '(' <formula w> '&' <formula w> ')';
<formula w> = 'A' x <formula x w>, x:<variable>;
<element w v u> = v, v:<variable>;
<variable> = 'x';
<variable> = 'y';
<variable> = 'z';
```

Let us provide examples of formulas generated and not generated by this grammar.

Positive examples, valid formulas generated by this grammar for the concept <formula ...> with different parameters. The examples are written in the form:

$$'example' :< formula\ 'parameters' >:$$

- $'P(x)' :< formula\ 'x' >$, an atomic formula with the free variable x,
- $'Q(x,y)' :< formula\ 'xy' >$, an atomic formula with two free variables,
- $'(P(x)\ \&\ P(y))' :< formula\ 'xy' >$, a conjunction of two atomic formulas,
- $'AxP(x)' :< formula >$, a closed formula with a universal quantifier,
- $'AxQ(x,y)' :< formula\ 'y' >$, a formula with a quantifier and a free variable y,
- $' - P(x)' :< formula\ 'x' >$, negation of an atomic formula,
- $'AxAxP(x)' :< formula >$, repeated binding of the same variable is allowed,
- $'AzQ(x,y)' :< formula\ 'xy' >$, the quantifier can bind a variable not present in the formula.

Negative examples (invalid formulas):

- $'P(w)'$, the variable w is not defined in the grammar,
- $'Q(x,x,y)'$, the predicate Q has only two arguments,
- $'P(x)\ \&\ Q(y)'$, mandatory parentheses around the conjunction are missing,
- $'P(x)' :< formula >$, the formula is not closed,
- $'P(x)' :< formula\ 'y' >$, the formula contains a variable not from the specified list.

§ 3. Special type of parameterized grammars for programming

3.1. Unambiguous parameterized grammars

A parameterized grammar is called unambiguous (functional) if the value of any expression of the form $< N\ e >$ consists of no more than one element. Such grammars define functions and can be used as programs to compute these functions. We will refer to these grammars as program-grammars.

One such class of grammars is described below. In the rules of this class, descriptions of permissible sets of variable values are absent. A single standard set of values <tree> (binary tree) is implied, with the following syntax (defined in Backus–Naur form, where terminal symbols are enclosed in apostrophes):

```
<tree> ::= '\' <character> | '(' <tree> ')' <tree>,
<character> - an arbitrary symbol from a pre-selected alphabet.
(a list of permissible terminal symbols for the considered class
of grammars)
```

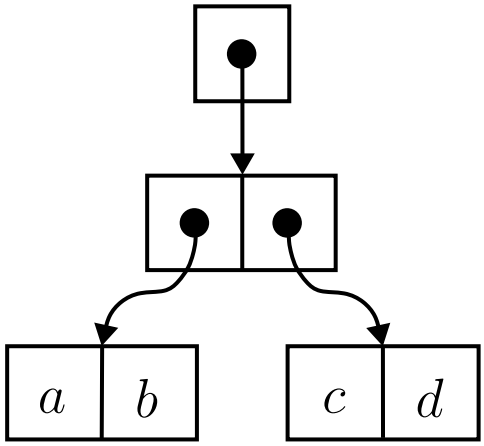

Fig. 1. Branched list representation. Example. List representation ((a)b)(c)d

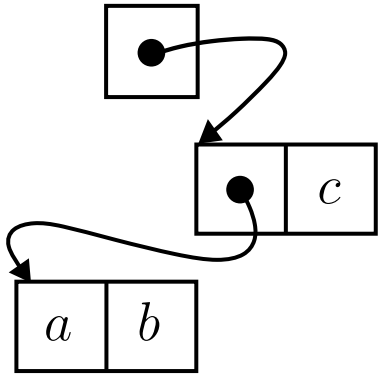

Fig. 2. Branched list representation. Example: Representation of the list $(\backslash a)\backslash b)\backslash c$

Here, the notation $\backslash c$ describes a tree consisting of a single leaf marked with the symbol c, and the notation $(a)b$ describes a complex tree consisting of left and right subtrees a and b, respectively. Non-leaf nodes are not marked, and edges are naturally considered to be labeled as "left" and "right". The asymmetric notation $(a)b$ is chosen for the convenience of representing more complex data structures (see below). The binary tree structure is chosen because it is fairly simple and allows for easy modeling of other structures.

Note that this is merely a representation of a tree as a chain of symbols. In practice, when working with such data, branched lists are typically used (see, for example, figures 1 and 2).

Ordinary one-dimensional chains of symbols can be easily modeled by such binary trees. For example, if we need to encode the chain "bac", we will use the tree $(\backslash a)(\backslash b)(\backslash c)$empty (see figure 3), where "empty" refers to an empty tree — a leaf marked with the symbol "space" ($\backslash\backslash$).

In future representations, such an empty tree will sometimes be omitted. If x is a tree of the form $(x_1)\ldots(x_n)$, it can be concatenated with any tree y. The result will be the tree $(x_1)\ldots(x_n)y$ (see figure 4), which is written as xy (the notation of two trees written consecutively implies the concatenation operation).

If x does not have such a form (i.e., does not end with an empty tree in the rightmost branch), then the operation xy is not defined.

As rules for unambiguous parameterized grammars, we consider only statements of the form:

$$< N\ t >= u;$$

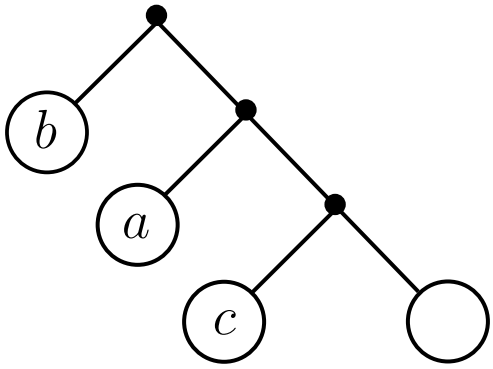

Fig. 3. Branched list representation. Example:Representation of the string “bac” = $(\backslash b)(\backslash a)(\backslash c)$

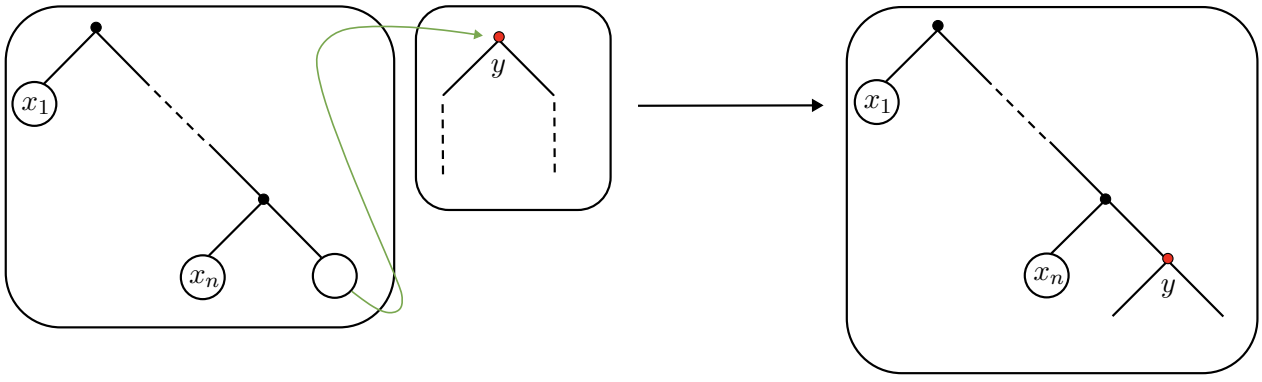

Fig. 4. Concatenation of trees

next we give a formal description of the class of unambiguous parameterized grammars.

3.2. Core concepts and structure of grammar-programs

The class of grammar-programs can be viewed as a special programming language, a tool for data processing based on manipulating binary trees. Grammar-programs are unambiguous parameterized grammars that operate on binary trees.

3.3. Formal syntax of the grammar-program language

3.3.1 Context-free metasyntax of grammar-programs

The original metasyntax of grammar-programs is defined as follows using Backus–Naur forms (lines marked with "–" are comments):

```
<program> ::= <rule> | <program> <rule>,
<rule> ::= '<' <name> <left> '>' '=' <right> ';',
<left> ::= '\' <character>
         | <name>
         | '(' <left> ')' <left>,
- comment: <left> is the "left part of the rule"
<right> ::= '\' <character>
          | <name>
          | '(' <right> ')' <right>
          | '<' <name> ' ' <right> '>',
- comment: <right> is the "right part of the rule",
- metaconcepts <name> and <character> are defined above.
```

As we can see, the syntax lacks concatenation of chains and non-terminal symbols in the left parts of rules, which simplifies the computation of the function defined by such a program. The specified binary trees can be efficiently represented in memory, allowing for an instantaneous transition from the left parenthesis to the right one.

3.3.2 Context conditions

It is required that in program-grammars, variables in the left part of the rule do not repeat. This is a requirement for implementation efficiency; otherwise, during execution, it would be necessary to compare the values of the variables. If comparison of values is needed, a special built-in function for comparing values will be provided in the programming system.

All variables in the right part of the rule (to determine their values) must appear in its left part. Variables in the right part can be repeated. One can assume that these values are then copied. However, in implementation, only references to them may be copied. It is also possible not to use some variables from the left part in the right part. In that case, it can be assumed that these values are destroyed. However, in implementation, only references to them may disappear. In case of necessity, a more or less efficient "garbage collection" will be provided. In some particularly efficient implementations, these permissions can also be replaced with corresponding prohibitions and implemented with special built-in functions for copying and destruction.

For unambiguity, all rules must be incompatible in their left parts (i.e., the rule that starts the execution of any function is determined unambiguously by the argument of that function). Two left parts are considered compatible if there exists a tree that can be matched with both of these parts. It is not difficult to formulate an algorithm that effectively recognizes the compatibility of left parts.

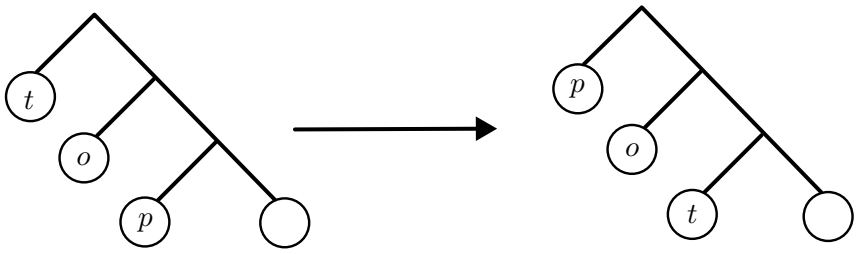

Fig. 5. Illustrating tree reversal (like "top"→ "pot")

3.4. Semantics of the programming language

3.4.1 Function definitions

One can consider that a grammar-program consists of a set of function definitions. Each function definition F includes the following:

– a list of rules of the form $< F\ L >= R$;

– where L represents the pattern of the function argument, the left part of the rule,

– R is the expression of the function value, the right part of the rule with possible calls to this and other functions.

3.4.2 Pattern matching mechanism

The process of executing a rule includes the following steps:

1) selecting values for the variables to match the left part to the function argument (context conditions are such that there is no more than one way to make such a selection),

2) substituting the found values into the right part,

3) computing the resulting expression.

If it was not possible to find a suitable rule, then the value of the function is undefined. It is also undefined if the described computation process does not terminate.

3.5. Examples

3.5.1 String reversal

A string reversal program might look as follows:

```
<reverse x> = <rev (x)\ >.
<rev ((x)y)z> = <rev (y)(x)z>.
<rev (\ )z> = z.
```

Example: when applying this program to the string "top" $(\backslash t)(\backslash o)(\backslash p)$, we get the string "pot" (see figure 5).

The process of this transformation can be depicted as follows:

```
<reverse (\t)(\o)(\p)\ > = <rev((\t)(\o)(\p)\ )\ >
- first rule applied,
= <rev((\o)(\p)\ )(\t)\ > = <rev((\p)\ )(\o)(\t)\ >
- second rule applied,
= <rev(\ )(\p)(\o)(\t)\ > = (\p)(\o)(\t)\
- finally, the third rule is applied.
```

3.5.2 String concatenation

Unlike many other languages where string and list concatenation (cat) is a built-in operation, in our language, it can be implemented through programming:

```
<cat((x)y)z>=(x)<cat(y)z>;
<cat()z>=z;
```

This program first outputs the elements of the first list and then appends the second list to them. This does not seem very efficient, so in practical implementation, such an operation could be built-in.

3.6. Example of a substantive grammar-program

Let us consider an example of solving the problem of simplifying a propositional formula with one variable. The task is to replace any propositional formula with one variable "Y" in the signature $-$ (negation), $\&$ (conjunction) with one of the equivalent formulas: 0 (false), 1 (true), Y, or N (negation of Y). The program with the function $< tr \dots >$, which solves this task, looks as follows:

```
<tr "Y"x> = "Y"x;
<tr "-"x> = <trnot <tr x>>;
<tr "("x> = <trand <tr x>>;
<trnot "Y"x> = "N"x;
<trnot "N"x> = "Y"x;
<trnot "0"x> = "1"x;
<trnot "1"x> = "0"x;
<trand (x)"&"y> = <conj (x)<tr y>>;
<conj (x)(y)")"z> = (<and (x)y>)z;
<and "0"x> = \0;
<and "1"x> = x;
<and "Y"\Y> = \Y;
<and "Y"\N> = \0;
<and "Y"\0> = \0;
<and "Y"\1> = \Y;
<and "N"\Y> = \0;
<and "N"\N> = \N;
<and "N"\0> = \0;
<and "N"\1> = \N;
```

For example, running the execution

```
<tr "(Y&-Y)">,
```

we obtain the following process:

```
<tr "(Y&-Y)"> =
<trand <tr "Y&-Y)">> =
<trand "Y&-Y)"> =
<conj "Y"<tr "-Y)">> =
<conj "Y"<trnot <tr "Y)">>> =
<conj "Y"<trnot "Y)">> =
<conj "YN)"> =
(<and "Y"\N>) = (\0) = "0".
```

§ 4. Built-in functions of the programming system

To process data, it is recommended to include the following functions in the programming system:

- Comparison function, for example:

```
    <comp(x)x> = \1;
    <comp(x)y> = \0;  - when x is not equal to y
```

- Functions for analyzing the type of character, for example:

```
    <letter x> = "1";  - if x is a letter
    <letter x> = "0";  - if x is not a letter
```

The programming system can also include functions for outputting strings to the output stream and reading strings from the input stream, as well as functions for file handling.

§ 5. Logical construction of program grammars from parameterized grammars (deductive programming)

For programming in grammars to work successfully, a transition is required from problem statements in terms of complex grammars to algorithms formulated as executable grammars (programs). This transition can be achieved through deductive programming, the logical derivation of programs in the form of functional grammars from problem statements in the form of general parameterized grammars. Below, we provide example of such deductive programming, the transformation of a parameterized grammar into a program grammar.

5.1. Example of deductive programming

5.1.1 Informal problem statement

It is required to create a program that translates a logical formula, written as a chain of symbols, into an internal representation in the form of a binary tree. For simplicity, we consider constant formulas of propositional logic with one logical connective “Sheffer stroke” and one logical constant 0 (false). Error handling is not considered for simplicity.

5.1.2 Formal problem statement

To clarify, we first present the grammar of the internal syntax of formulas <form>:

```
<form> = \0;
- case of the constant "false"
<form> = (<form>)<form>;
- case of a complex formula (the Sheffer stroke subsumed)
```

Now, we define the rules of the parameterized grammar establishing correspondence

```
<frml f> = t
```

between the internal

```
f:<form>
```

and external (string t) representations of formulas.

Since the definition of $form$ has two cases, the corresponding two cases also apply to the definition of $frml$:

```
<frml \0> = "0";
- case when the formula is the constant "false"
<frml (g) h> = "(" <frml g> "|" <frml h> ")";
- case when the formula is obtained by the Sheffer stroke,
no restrictions on g and h
```

The specification of the target function for converting the external representation of the formula into its internal representation (f) will be as follows:

```
<ana <frml f> x> = (f) x;
```

Note that it is not just a formula being processed, but text starting with a formula from which this formula is extracted in its internal form. This is done to simplify the problem being solved, as recognizing a formula at the beginning of the text is often required during its recursive resolution.

5.1.3 Solution to the problem: building the target function program

This section presents reasoning showing how, in our example, a program grammar can be constructed from a general parameterized grammar using logical construction. The generated sentences of the program grammar are marked with a "!" at the beginning of the line. Collecting them together, we obtain the generated program.

Consider the specification rule of our task:

```
<ana <frml f> x> = (f) x;
```

First, we analyze the cases of the term

```
<frml f>.
```

for each case of the definition $frml$, we generate the corresponding program rules:

– in the case

```
<frml \0> = "0";
```

that is, when $f = \backslash 0$, our specification rule turns into the following form:

```
!<ana "0" x> = (\0) x;
```

this is already a program grammar rule, further transformation is not required.

– In the case

```
<frml (g) h> = "(" <frml g> "|" <frml h> ")";
```

that is, when $f = (g)h$, the specification rule turns into the following form:

```
<ana "(" <frml g> "|" <frml h> ")" x> = ((g) h) x;
```

Next, to build the next program rule, we denote the expression

```
<frml g> "|" <frml h> ")" x
```

as i:

```
i = <frml g> "|" <frml h> ")" x
```

Thus, the next action of the program can start with

```
<ana "(" i> =...,
```

since the value of i starts with

```
$<frml g>$.
```

It is proposed to continue working with recursive application of ana, then the next action can take the program grammar rule:

```
!<ana "(" i> = <anaa <ana i>>;
```

where $anaa$ is the function that continues the computation of ana.

Note that

```
<ana i> = <ana <frml g> "|" <frml h> ")" x> = (g) "|" <frml h> ")" x,
```

from which we obtain the following specification for the function $anaa$:

```
<anaa (g) "|" <frml h> ")" x> = ((g) h) x;
```

Let's take a new variable j defined by the equality

```
j=<frml h> ")" x.
```

Thus, the next action of the program can start with

```
<anaa (g) "|" j> =...
```

Since the value of j starts with

```
<frml h>,
```

it is also proposed to continue working with recursive application of ana. The action is extracted as a program rule:

```
!<anaa (g) "|" j> = <anab (g) <ana j>>;
```

where $anab$ is the function that continues the computation of $anaa$.

Note that

```
<ana j> = <ana <frml h> ")" x> = (h) ")" x.
```

From this, we obtain the following specification for the function $anab$:

```
!<anab (g) (h) ")" x> = ((g) h) x;
```

This is already a program rule, further transformation is not required. There are no remaining rules that are not program rules. Thus, the problem of building the program is solved. To justify the correctness of the constructed program, it can also be shown that the size of the data processed by it decreases at each step. The final solution looks as follows:

```
!<ana "0" x> = (\0) x;
!<ana "(" i> = <anaa <ana i>>;
!<anaa (g) "|" j> = <anab (g) <ana j>>;
!<anab (g) (h) ")" x> = ((g) h) x;
```

§ 6. Conclusion

This work presents and analyzes a new approach to programming based on the use of parameterized grammars. This approach combines classical formal language theory with modern programming methods, creating a powerful and flexible tool for solving a wide range of tasks in computational linguistics and data processing.

The main results of the work can be summarized as follows.

1. A theoretical foundation for parameterized grammars over binary trees has been developed, generalizing classical concepts from formal language theory. It has been shown that grammars over linear chains are a special case of grammars over binary trees, allowing for a unified approach to solving various classes of problems.

2. A special class of unambiguous (functional) parameterized grammars has been proposed, which can be directly used as programs. These grammar-programs possess clear semantics and efficient implementation, making them practical programming tools.

3. Deductive synthesis of programs has been demonstrated, allowing for generation of efficient grammar-programs from task specifications written in the form of general parameterized grammars. This approach ensures the correctness of the obtained solutions.

4. The practical applicability of the proposed approach has been demonstrated through specific examples, including tasks related to processing logical formulas, string transformations, and conversions between various data representations. The developed formalism allows for the natural expression and efficient resolution of a wide class of symbolic computation problems.

The theoretical significance of the work lies in the advancement of formal language theory and programming methods. The proposed formalism of parameterized grammars over binary trees creates a new programming paradigm that combines the advantages of declarative and functional approaches. The method of deductive synthesis of programs opens up prospects for automating the programming process and verifying programs.

The practical value of the research consists in the creation of specific tools for solving language processing and symbolic computation tasks. The developed approach can find applications in the following areas:

- Compiler and language processor development
- Natural language processing systems creation
- Symbolic computation systems construction
- Development of program analysis and transformation tools
- Creation of specialized programming languages

Prospects for further research include:

1. Development of the theory of parameterized grammars, exploring their expressive power and computational complexity.

2. Development of efficient methods for implementing grammar-programs on various platforms.

3. Creation of tools for automatic program synthesis from specifications.

4. Integration of the proposed approach with machine learning methods.

5. Expansion of the application area to new classes of problems.

The proposed approach opens new horizons in programming language tasks, creating a theoretical and practical basis for the development of more efficient and reliable methods for processing formal and natural languages. The synthesis of classical ideas from formal language theory with modern programming methods creates a powerful toolkit for addressing current challenges in computational linguistics and artificial intelligence.

Implementing the system by converting it into modern programming languages seems to be the most promising approach. This not only simplify the implementation of the system into

existing projects, but also opens up access to the extensive libraries of these languages, especially in the field of machine learning and data processing. This approach avoids the problems faced by earlier systems that attempted to create a completely autonomous execution environment.

Further development of the system in this regard is seen in the creation of effective converters into various programming languages and the development of specialized interfaces for interaction with modern machine learning and data analysis libraries. This will allow preserving the advantages of the formal approach when working with grammars, while simultaneously using the full power of modern development tools.

The key advantages of this approach are as follows.

1. Practicality of implementation:
– use of existing optimized programming systems,
– access to debugging and profiling systems,
– ease of integration into existing projects.
2. Extensibility:
– easy addition of new functionality through libraries,
– ability to use modern development tools,
– access to language ecosystems.
3. Prospective:
– natural integration with machine learning tools,
– possibility of parallel computing,
– access to cloud technologies.

REFERENCES

1. Chomsky N. On certain formal properties of grammars, *Information and Control*, 1959, vol. 2, issue 2, pp. 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6
2. Backus J. W. The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference, *ICIP Proceedings*, 1959, pp. 125–132. https://cir.nii.ac.jp/crid/1572824501224489728
3. Knuth D. E. Semantics of context-free languages, *Mathematical Systems Theory*, 1968, vol. 2, no. 2, pp. 127–145. https://doi.org/10.1007/BF01692511
4. Hehner E. C. R., Silverberg B. A. Programming with grammars: an exercise in methodology-directed language design, *The Computer Journal*, 1983, vol. 26, issue 3, pp. 277–281. https://doi.org/10.1093/comjnl/26.3.277
5. Paakki J. Attribute grammar paradigms — a high-level methodology in language implementation, *ACM Computing Surveys (CSUR)*, 1995, vol. 27, issue 2, pp. 196–255. https://doi.org/10.1145/210376.197409
6. Griswold R. E. A history of the SNOBOL programming languages, *ACM Sigplan Notices*, 1978, vol. 13, issue 8, pp. 275–308. https://doi.org/10.1145/960118.808393
7. Turchin V. F. *Refal-5 programming guide and reference manual*, Holyoke: New England Publishing Co., 1989.
8. Turchin V. F. The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1986, vol. 8, issue 3, pp. 292–325. https://doi.org/10.1145/5956.5957
9. Nemytykh A. P. The supercompiler SCP4: general structure, *Perspectives of System Informatics*, Berlin–Heidelberg: Springer, 2004, pp. 162–170. https://doi.org/10.1007/978-3-540-39866-0_18
10. Klimov A. V. A Java supercompiler and its application to verification of cache-coherence protocols, *Perspectives of System Informatics*, Berlin–Heidelberg: Springer, 2009, pp. 185–192. https://doi.org/10.1007/978-3-642-11486-1_16
11. Romanenko A. Inversion and metacomputation, *ACM SIGPLAN Notices*, 1991, vol. 26, issue 9, pp. 12–22. https://doi.org/10.1145/115865.115868

12. Besova G., Steenken D., Wehrheim H. Grammar-based model transformations, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Volume 2*, IEEE, 2014, pp. 1601–1610. https://doi.org/10.15439/2014F144
13. Carette J., Kiselyov O. Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code, *Science of Computer Programming*, 2011, vol. 76, issue 5, pp. 349–375. https://doi.org/10.1016/j.scico.2008.09.008

Received 15.01.2025
Accepted 23.04.2025

Milad Joudakizadeh, Post-Graduate Student, Department of Computing Technologies and Intellectual Systems of Big Data, Udmurt State University, ul. Universitetskaya, 1, Izhevsk, 426034, Russia.
ORCID: https://orcid.org/0000-0002-6167-6237
E-mail: dzhudakizade@udsu.ru

Anatoly Petrovich Beltiukov, Doctor of Physics and Mathematics, Professor, Department of Computing Technologies and Intellectual Systems of Big Data, Udmurt State University, ul. Universitetskaya, 1, Izhevsk, 426034, Russia.
ORCID: https://orcid.org/0000-0002-3433-9067
E-mail: belt.udsu@mail.ru

Citation: M. Joudakizadeh, A. P. Beltiukov. Programming in grammars, *Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki*, 2025, vol. 35, issue 2, pp. 315–334.

КОМПЬЮТЕРНЫЕ НАУКИ 2025. Т. 35. Вып. 2. С. 315–334.

М. Джудакизаде, А. П. Бельтюков

Программирование в грамматиках

Ключевые слова: атрибутные грамматики, параметрические грамматики, параметризованные грамматики, языковое программирование, символьные преобразования, искусственный интеллект, машинное обучение.

УДК 004.432.42

DOI: 10.35634/vm250210

В данной работе рассматривается подход к программированию, основанный на использовании параметризованных грамматик. Понятия этих грамматик снабжены параметрами, которые также могут быть объектами грамматик. Такие грамматики являются довольно мощным инструментом, их предлагается использовать для формулирования постановок задач преобразования лингвистических данных. Эти грамматики можно использовать непосредственно для обработки информации, но это может оказаться не эффективным. Поэтому выделяется специальный эффективный в применении класс таких грамматик. Предлагается специальная система однозначных (функциональных) параметризованных грамматик, которую можно использовать как эффективный язык программирования лингвистических задач. Описываются идеи дедуктивного синтеза программ в этой системе из постановок задач в общих параметризованных грамматиках с помощью логического вывода с перспективой последующей автоматизации. Демонстрируется практическое применение предложенного инструмента на примерах обработки логических формул и решения других задач. Эта работа продолжает идеи Валентина Турчина в области языка РЕФАЛ.

СПИСОК ЛИТЕРАТУРЫ

1. Chomsky N. On certain formal properties of grammars // Information and Control. 1959. Vol. 2. Issue 2. P. 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6
2. Backus J. W. The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference // ICIP Proceedings. 1959. P. 125–132. https://cir.nii.ac.jp/crid/1572824501224489728
3. Knuth D. E. Semantics of context-free languages // Mathematical Systems Theory. 1968. Vol. 2. No. 2. P. 127–145. https://doi.org/10.1007/BF01692511
4. Hehner E. C. R., Silverberg B. A. Programming with grammars: an exercise in methodology-directed language design // The Computer Journal. 1983. Vol. 26. Issue 3. P. 277–281. https://doi.org/10.1093/comjnl/26.3.277
5. Paakki J. Attribute grammar paradigms — a high-level methodology in language implementation // ACM Computing Surveys (CSUR). 1995. Vol. 27. Issue 2. P. 196–255. https://doi.org/10.1145/210376.197409
6. Griswold R. E. A history of the SNOBOL programming languages // ACM Sigplan Notices. 1978. Vol. 13. Issue 8. P. 275–308. https://doi.org/10.1145/960118.808393
7. Turchin V. F. Refal-5 programming guide and reference manual. Holyoke: New England Publishing Co., 1989.
8. Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems (TOPLAS). 1986. Vol. 8. Issue 3. P. 292–325. https://doi.org/10.1145/5956.5957
9. Nemytykh A. P. The supercompiler SCP4: general structure // Perspectives of System Informatics. Berlin–Heidelberg: Springer, 2004. P. 162–170. https://doi.org/10.1007/978-3-540-39866-0_18
10. Klimov A. V. A Java supercompiler and its application to verification of cache-coherence protocols // Perspectives of System Informatics. Berlin–Heidelberg: Springer, 2009. P. 185–192. https://doi.org/10.1007/978-3-642-11486-1_16
11. Romanenko A. Inversion and metacomputation // ACM SIGPLAN Notices. 1991. Vol. 26. Issue 9. P. 12–22. https://doi.org/10.1145/115865.115868

12. Besova G., Steenken D., Wehrheim H. Grammar-based model transformations // Proceedings of the 2014 Federated Conference on Computer Science and Information Systems. Volume 2. IEEE, 2014. P. 1601–1610. https://doi.org/10.15439/2014F144
13. Carette J., Kiselyov O. Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code // Science of Computer Programming. 2011. Vol. 76. Issue 5. P. 349–375. https://doi.org/10.1016/j.scico.2008.09.008

Поступила в редакцию 15.01.2025
Принята к публикации 23.04.2025

Джудакизаде Милад, аспирант, кафедра вычислительных технологий и интеллектуальных систем больших данных, Удмуртский государственный университет, 426034, Россия, г. Ижевск, ул. Университетская, 1.
ORCID: https://orcid.org/0000-0002-6167-6237
E-mail: dzhudakizade@udsu.ru

Бельтюков Анатолий Петрович, д. ф.-м. н., профессор, кафедра вычислительных технологий и интеллектуальных систем больших данных, Удмуртский государственный университет, 426034, Россия, г. Ижевск, ул. Университетская, 1.
ORCID: https://orcid.org/0000-0002-3433-9067
E-mail: belt.udsu@mail.ru

Цитирование: М. Джудакизаде, А. П. Бельтюков. Программирование в грамматиках // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. 2025. Т. 35. Вып. 2. С. 315–334.