A Templated C++ Parser Library for Large Language Model Workflows

Remi Savchuk
HSE University
Moscow, Russia
ORCID: 0009-0007-9083-0292

Ekaterina Trofimova
HSE University
Moscow, Russia
Email: etrofimova@hse.ru

***Abstract*—With the adoption of structured grammars in LLM workflows, the limitations of existing parser generators are hindering their effective use with LLMs. Multi-stage parser generators with complex pipelines are too complicated to deploy in HPC environments, and the natural language that LLMs are trained to output is not always context-free. In this paper, we formulate the limitations of existing parsers, introduce a novel context-aware parsing algorithm, and present a C++ parser generator library [1] for LLM-specific tasks. Furthermore, a method to iterate over a superset of grammars at runtime is presented.**

I. Introduction

Parser generators have been used in various tasks since their inception in the 1970s [1]–[3]. They generate parsers automatically, which is essential for parsing programming languages and structured text. Recursive descent parsers are the easiest to implement [4], although they have limitations on the type of recursive grammars that can be parsed by them [5]. Shift-reduce parsers, however, read the grammar from the leaf nodes and can therefore handle much more complex grammars.

More recently, the emergence of large language models (LLMs) has disrupted many traditional workflows in software engineering. Tasks such as code synthesis, automated program repair, and DSL generation are increasingly powered by LLMs, which frequently produce structured output that mimics formal grammars [6]–[8]. However, integrating these outputs with existing parser infrastructures remains a significant challenge.

Most parser generators were designed with hand-written grammars and context-free language assumptions in mind. In contrast, LLM-generated text is probabilistic, sometimes loosely structured, and often context-sensitive. This mismatch creates friction in applications where the output of an LLM must be parsed, validated, or executed reliably — especially in environments that require high-throughput and robustness, such as agent pipelines, high-performance computing (HPC), or automated code reviews.

A. Motivation

In large language models, parsers are employed to constrain the set of possible tokens, enabling the generation of formal grammars. In the original paper [9], this was achieved by checking partial string matches using the Grammatical Framework. In subsequent articles presenting enhanced decoder versions, such as SynCode [10], a bottom-up LR parsing technique is utilized for partial decoding. However, this method of structural decoding, which merely masks incorrect tokens without providing the model with information about the grammar, can result in various token generation artifacts, including the empty string problem and hallucinations. The empty string problem occurs when a whitespace character or an end-of-sequence (EOS) token has the highest probability after masking non-fitting tokens. These issues are particularly pronounced in grammars that are substantially different from natural language, such as JSON or HTML, which have specific terminal sequences. Adding samples of the target grammar to the training data does not fully address the problem, since natural language presents a greater portion of the training data, and token probabilities may not be changed to a substantial degree in order to prevent the empty string problem.

Alternatively, the empty string and hallucination problems could be mitigated by using grammars that are closer in terminal sequence structure to natural language. For example, the grammar could be designed without special tokens at the start of nonterminals, making the presence of certain alphanumeric terminals context-dependent. This approach would allow the model to have fewer constraints on its output and to better understand the structure of the grammar, as it aligns more closely with the natural languages present in the training data. Information about specific grammar rules can still be provided to the model in the prompt. While this method relies less on constrained decoding, it necessitates the use of a parser generator that supports context-dependent grammars. Although such grammars can still be described in the Extended Backus-Naur Form (EBNF), static parsers cannot utilize context during parsing.

Another approach involves permuting over possible grammar configurations to find the one that performs best with a particular model. This can be achieved by generating parsers at runtime during model evaluation, as compiling all versions of parsers in advance would require building 2^n different parser versions. Native parser generators, which require only a single compilation step, are easier to deploy in high-performance computing (HPC) environments compared to parsers with multiple compilation stages. Additionally, grammar serialization, which is necessary for formal grammar rule conversion

[1]The code is available via the link: https://github.com/enaix/SuperCFG/

between pipeline steps, is a crucial feature. Parser generators, constrained decoding frameworks, and other tools often utilize specific grammar definition formats.

B. Static vs. dynamic parsing

Static parser generators analyze the grammar during the compilation step and yield a set of deterministic parser states and transitions, typically encoded as parsing tables. This approach was pioneered by Knuth through the introduction of LR parsing [11], which forms the basis for many widely used shift-reduce algorithms. While classic LR parsers generally permit only a single valid state transition at each step, more flexible algorithms such as Earley's parser allow for the exploration of multiple ambiguous transitions during parsing [12]. However, even these extensions remain limited in their ability to adapt dynamically to runtime context or probabilistic input streams.

Dynamic parser generators, by contrast, do not have predefined parser states and transitions. These systems attempt to match token sequences to grammar rules on-the-fly, enabling richer context analysis and dynamic ambiguity handling without backtracking. Since static parsing algorithms do not operate on the stack as a whole, but rather on a predefined state, it is problematic to implement the handling of ambiguities [13]. Recursive-descent parsers represent a basic form of this approach, but more sophisticated dynamic parsers can implement bottom-up parsing, predictive lookahead, or other techniques.

C. Contributions

In this paper, we introduce a dynamic shift-reduce parser generator tailored to the needs of LLM-driven software engineering. Our contributions can be summarized as following:

- A novel dynamic shift-reduce parsing algorithm that supports context-sensitive grammars and enables grammar-aware behavior during runtime.
- A header-only templated C++ parser generator library that implements the parsing algorithm, supports grammar serialization and is suitable for runtime execution.
- A novel method for iterating over possible grammar sets for optimizing LLM-constrained decoding, with seamless integration into ML frameworks through just-in-time compilation and compile-time grammar serialization.

II. Methodology

A. Parser structure

Like in classical static shift-reduce parsers, main dynamic shift-reduce parser routine has shift and reduce routines shown in Fig. 1. Shift routine takes a tokenized string $\mathfrak{T}$ and adds a token to the stack $\mathcal{S}$. Shift routine then calls the reduce routine, which iterates over the stack, finds potential nonterminal candidates and calls the descend routine to the each match. Descend routine tries to perform a full match of the current stack against the rule candidate. These operations form the base of the algorithm, and additional operations may be performed in the reduce routine to eliminate conflicts and better analyze the context.

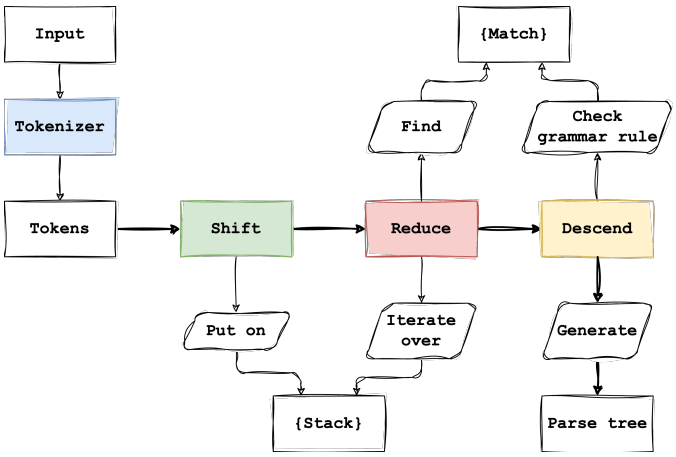

Fig. 1. Dynamic shift-reduce algorithm pipeline

B. Shift and reduce routines

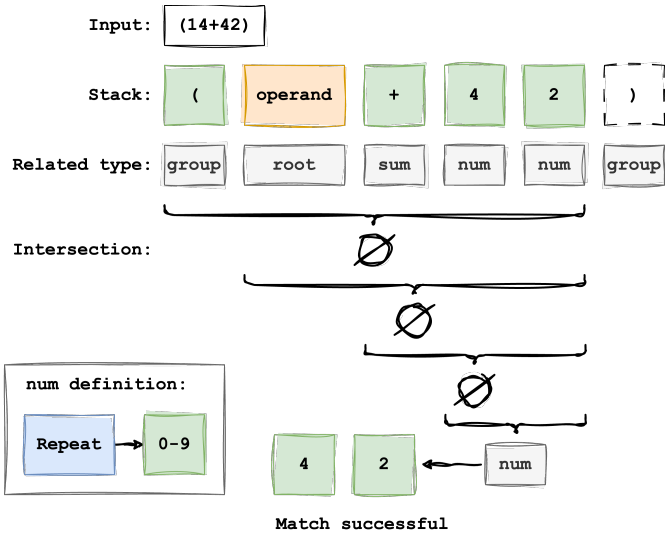

Fig. 2. Example of a reduce routine at some stack state for a calculator grammar

Algorithm 1 defines the shift routine, which puts tokens $\tau \in \mathfrak{T}$ to the stack $S : \{\tau_0, \ldots, \tau_n\}$. Each token τ is represented either by a terminal T or a nonterminal NT. Then, the reduce routine defined in the Algorithm 2 iterates over the stack S, converts each terminal $T \in \mathcal{S}$ into the set of rules which contain T $\mathfrak{R}(T) : \{NT_0, \ldots, NT_k\}$ and each nonterminal $NT \in \mathcal{S}$ into the set of related rules which also contain nonterminal NT $\mathfrak{RT}(NT) : \{NT_0, \ldots, NT_m\}$. An example of this algorithm is shown in Fig. 2. **lookahead** routine performs a simple one symbol lookahead with the help of a *FOLLOW* set. **check_ctx** routine is a placeholder for the context analysis module. Both of these routines are optional.

Descend routine iterates over the grammar rule and matches each token against the grammar rules. The behavior of this routine is similar to the recursive descent, but it does not descend into other rule definitions.

```
fn shift(&node, root, 𝔗) → bool:
    𝒮 ← 𝔗[0];
    i_t ← 1;
    while do
        if ¬reduce(𝒮, node, 𝔗, i_t) then
            if i_t = |tokens| then
                break;
            end
            𝒮 ← 𝔗[i_t];
            i_t ← i_t + 1;
        end
    end
    return |𝒮| = 1 && 𝒮[0] = root ;
```

Algorithm 1: Shift routine algorithm

```
fn reduce(𝒮, &node, 𝔗, i_t) → bool:
    for ( i := 0; i < |stack|; i ← i + 1 ) {
        λ_R ← λx. if x is T then ℜ(x) else ℜ𝔗(x)
         end;
        ;          /* Find the intersection
         between related types */
        ℐ ← λ_R(S[i]) ∩ ··· ∩ λ_R(S[n − 1]);
        for ( match ← ℐ ) {
            if ¬lookahead(match, 𝔗, i_t) then
                continue;
            end
            j ← 0 ;
            ok ← descend(𝒮, i, match . rule, &j) ;
            if ok && i + j = n then
                if ¬check_ctx(...) then
                    continue;
                end
                ... ;   /* Populate the AST */
                𝒮 ← {𝒮[0], ..., 𝒮[i − 1], match};
                return true;
            end
        }
    }
    return false;
```

Algorithm 2: Reduce routine algorithm

C. Heuristic conflicts and ambiguities resolver

The base version of the parser algorithm defined above can be expanded with modules that analyze context and make decision whether to accept or reject the match candidate. The way these modules operate make them heuristic in the sense that they analyze the grammar first and then take educated guesses about a match candidate during runtime. This allows the parser algorithm to be expanded in the future, with more advanced context analysis algorithms being able to parse larger sets of grammars.

D. Reducibility checker

Reducibility checker $\mathcal{RC}(1)$ is a module that checks if a match candidate can be reduced one step ahead in the future. During compilation $\mathcal{RC}(1)$ builds a mapping between a nonterminal and its minimal position inside of each rule definition.

$$\mathcal{P}_{\mathcal{RC}(1)} : |NT| \to (\mathbb{R}, |NT|)^N$$
$$\mathcal{P}_{\mathcal{RC}(1)} : NT \mapsto \{\{i_0, NT_0\}, \ldots, \{i_N, NT_N\}\}$$

This allows the algorithm to perform partial descend routines for ensuring that the symbol can be reduced one step ahead in the future. Furthermore, this checks allows the checker to approximate the current state of the stack context, which can be defined as the level of nesting of each rule. The behavior of $\mathcal{RC}(1)$ is described in Algorithm 3. This algorithm can theoretically be executed multiple times, which would ensure that the match candidate can be reduced k steps into the future.

```
fn can_reduce(𝒮, match) → bool:
    for ( i, rule ← 𝒫_{𝒞(1)} ) {
        if i > |𝒮| then
            continue;
        end
        j ← |𝒮| − i − 1;
        k ← 0;
        ;     /* Here 𝒮 contains the match
         candidate */
        ok ← descend(𝒮, j, rule, &k);
        if k ≥ i then
            ctx[rule] ← ctx[rule] + 1 ;
            return true;
        end
        if ctx[rule] > 0 then
            return true;
        end
        return false;
    }
    return true;
    ;     /* ctx[rule] is decremented in the
     reduce routine */
```

Algorithm 3: Reducibility checker algorithm

E. Prefix-based context analyzer

In order to better analyze the context, an improved version of the reducibility checker could be made. This algorithm uses prefixes and postfixes of a rule in order to analyze current context, which allows the algorithm to accept or reject candidates based on the current stack context. If a nonterminal NT is not included in the recursive set of related rules $\mathfrak{RT}(\mathfrak{RT}(\ldots(\mathfrak{RT}(rule))))$, then this element cannot be inside of this rule, and the match candidate should be discarded.

The algorithm builds a set of prefixes and postfixes the same way as $\mathcal{RC}(1)$, but ambiguous element positions are not considered. At runtime, the algorithm constructs two sets $PrefixTODO$ and $PostfixTODO$, which contain stack indices with ambiguous context. Presence of an ambiguity

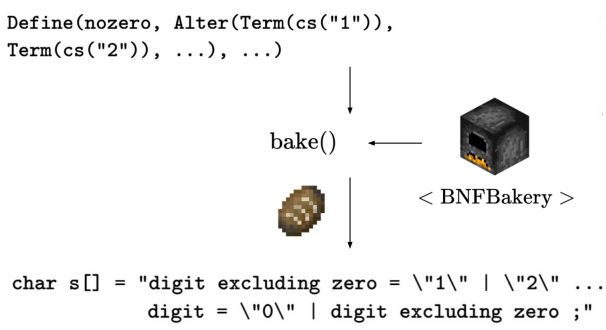

Fig. 3. Compile-time serialization process: abstract grammar definitions are transformed by the baking function into a custom EBNF-like notation suitable for downstream use.

indicates that a reduction cannot be performed, and more tokens should be analyzed by the module in order to correctly identify the current context. This algorithm allows the parser to correctly handle ambiguous rules with terminals dependent on the context. As a practical example, the parser could handle certain cases of markup languages with unquoted tags or unescaped character sequences.

F. Grammar serialization

A distinctive feature of our system is the introduction of the *baking function* $\mathscr{B}$ 1, which performs compile-time serialization of grammars into a custom EBNF-like textual representation. This mechanism, illustrated in Fig. 3, is essential for integrating symbolic grammar definitions with external components such as language models.

In practice, $\mathscr{B}$ transforms an internal abstract representation of the grammar to a well-formed output, useful for tools like `llama.cpp` to perform bounded or grammar-constrained decoding. The mechanism is capable of facilitating the integration among parsing infrastructure and language model inference to be close to consistency and reduce runtime overhead.

The serialization format is readable and rich in structure, preserving the full expressiveness of the underlying grammar, e.g., operator precedence, associativity, and optional constructs. Since the operation is performed at compile time, it has strong type guarantees, improved performance, and seamless metaprogramming integration. This shift opens the door to a variety of DSL-based workflows, such as code generation, grammar-constrained language modeling, and hybrid symbolic-neural processing pipelines.

$$\begin{aligned} \mathscr{B} &: |G| \to \mathbb{R}^M \\ \mathscr{B} &: (\mathcal{N}, \Sigma, \mathcal{P}, \mathcal{R}) \mapsto X^{BNF} \end{aligned} \tag{1}$$

G. Iteration over grammars

We introduce a method for iterating over a set of possible grammars $\mathfrak{G}$. This method allows the user to solve tasks which require finding the most optimal grammar. Such method is needed for finding a grammar that performs the best for a particular task. In order to generate a grammar and its parser at runtime, we may use C++ *Cling* [14] just-in-time compiler. This tool allows us to define a C++ source file containing grammar and execute it on-the-spot without intermediate steps, which is an important factor when launching code in restricted environments. Grammar serialization is essential for passing current grammar configuration into external machine learning frameworks, which often use custom grammar forms like GBNF in *llama.cpp* [15].

III. Experiments and results

TABLE I
Basic operator types

Name	Description	Example
Concat	Concatenation	A, B, C
Alter	Alternation	A or B or C
Define	Definition	$A := \ldots$
Optional	None or once	$[A]$
Repeat	None or $+\infty$ times	$\{A\}$
Group	Parenthesis	(A)
Comment		$(*ABC*)$
SpecialSeq	Special sequence	$?ABC?$
Except	Exception	$A - B$
End	Termination	$ABC.$
RulesDef	Rules definition	

TABLE II
Extended operator types

Name	Description
RepeatExact	Repeat exactly M times
RepeatGE	Repeat at least M times
RepeatRange	Repeat $[M, N]$ times

We present a toy grammar for an arithmetic calculator without operator precedence. The rules are expressed in EBNF format as follows:

```
number     = { digit } ;
add        = op, "+", op ;
sub        = op, "-", op ;
mul        = op, "*", op ;
div        = op, "/", op ;
arithmetic = add | sub | mul | div ;
op         = number | arithmetic | group ;
group      = "(", op, ")" ;
```

The same rules are implemented in C++ using a domain-specific language (Table I, II), which is defined in Appendix A.

To evaluate the grammar, we parse the input string `12*(3+42)` using a parser with enabled lookahead. The resulting parse tree is shown in Fig. 4.

Additionally, a JSON-like grammar example can be found in the source code of the library.

A. Complexity analysis

As the parser algorithm is dynamic, it performs much slower than the linear-time static shift-reduce parsers. The shift

```
parser output :
 (1 elems) :
|  op (1 elems) :
|  |  arithmetic (1 elems) :
|  |  |  mul (2 elems) : *
|  |  |  |  op (1 elems) :
|  |  |  |  |  group (1 elems) : ()
|  |  |  |  |  |  op (1 elems) :
|  |  |  |  |  |  |  arithmetic (1 elems) :
|  |  |  |  |  |  |  |  add (2 elems) : +
|  |  |  |  |  |  |  |  |  op (1 elems) :
|  |  |  |  |  |  |  |  |  |  number (1 elems) :
|  |  |  |  |  |  |  |  |  |  |  digit (0 elems) : 42
|  |  |  |  |  |  |  |  |  op (1 elems) :
|  |  |  |  |  |  |  |  |  |  number (1 elems) :
|  |  |  |  |  |  |  |  |  |  |  digit (0 elems) : 3
|  |  |  |  op (1 elems) :
|  |  |  |  |  number (1 elems) :
|  |  |  |  |  |  digit (0 elems) : 12
main() : passed
```

Fig. 4. Parse tree for the input `12*(3+42)` using the toy arithmetic grammar

routine pushes n tokens from the input string to the stack $\mathcal{S}$ and performs at most h reduce operations, where h is the maximum depth of the grammar, which makes the reduce routine linear in $\mathcal{O}(n)$.

Reduce routine needs to perform an $\mathcal{O}(|\mathcal{S}|^2)$ iteration over the stack $\mathcal{S}$, As there are at most $|G|$ rules in the grammar, there are at least $|G|$ possible match candidates, so the complexity of the reduce routine is $\mathcal{O}(|\mathcal{S}|^2)$. The descend routine has the complexity of $\mathcal{O}(\max\{|rule| \; | rule \in G\}) = \mathcal{O}(1)$, and both the reducibility checker and the context analyzer complexity are also dependent only on the grammar.

This makes the parser algorithm complexity equal to $\mathcal{O}(n \cdot |\mathcal{S}|^2 \cdot |G|) = \mathcal{O}(n^3)$, with $|G|$ being the grammar size, which is not dependent on the input. While the worst-case complexity of the algorithm is equal to $\mathcal{O}(n^3)$, the complexity is limited by the size of the stack, which is predictable based on the grammar properties. With the optimal grammar rules and parser configuration the stack size can be kept relatively small, which makes the algorithm perform closer to a linear time.

B. Future work

As of the time of writing, the prefix-based context analyzer has not yet been fully implemented. This limitation restricts our ability to perform a comprehensive comparative analysis against existing algorithms. Future work will focus on finishing the context analyzer to improve reduce-reduce conflicts resolving and performing the full evaluation of the algorithm.

IV. Conclusion

In this paper, we have presented a new dynamic shift-reduce parser algorithm. Our proposed approach leverages templated C++ facilities to enable efficient and flexible parser generation without relying on external tools. By adding a bake function for compile-time serialization to an EBNF-like representation, we provide a simple integration of symbolic grammar definitions and parsing infrastructure.

References

[1] S. D. Swierstra and P. R. Azero Alcocer, "Fast, error correcting parser combinators: A short tutorial," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 1999, pp. 112–131.

[2] N. A. Klyucherev and I. I. Rastvorova, "A simple run-time parser generator," in *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE, 2021, pp. 458–463.

[3] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Garcia, "An empirical evaluation of lex/yacc and antlr parser generation tools," *Plos one*, vol. 17, no. 3, p. e0264326, 2022.

[4] S. Purve, P. Gogte, and R. Bhave, "Stack and queue based parser approach for parsing expression grammar," in *2023 11th International Conference on Emerging Trends in Engineering & Technology-Signal and Information Processing (ICETET-SIP)*. IEEE, 2023, pp. 1–6.

[5] A. Wöß, M. Löberbauer, and H. Mössenböck, "Compiler generation tools for c#," *IEE Proceedings-Software*, vol. 150, no. 5, pp. 323–327, 2003.

[6] M. Setak and P. Madani, "Fine-tuning llms for code mutation: A new era of cyber threats," in *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*. IEEE, 2024, pp. 313–321.

[7] E. Trofimova, E. Sataev, and A. Ustyuzhanin, "Linguacodus: a synergistic framework for transformative code generation in machine learning pipelines," *PeerJ Computer Science*, vol. 10, p. e2328, 2024.

[8] M. A. Haque, "Llms: A game-changer for software engineers?" *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, p. 100204, 2025.

[9] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-constrained decoding for structured nlp tasks without finetuning," *arXiv preprint arXiv:2305.13971*, 2023.

[10] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, "Syncode: Llm generation with grammar augmentation," *arXiv preprint arXiv:2403.01632*, 2024.

[11] D. E. Knuth, "On the translation of languages from left to right," *Information and control*, vol. 8, no. 6, pp. 607–639, 1965.

[12] J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[13] A. Aho, S. Johnson, and J. Ullman, "Deterministic parsing of ambiguous grammars," *Communications of the ACM*, vol. 18, pp. 441–452, 01 1973.

[14] V. Vassilev, P. Canal, A. Naumann, L. Moneta, and P. Russo, "Cling – the new interactive interpreter for ROOT 6."

[15] ggerganov. llama.cpp: Llm inference in c/c++. [Online]. Available: https://github.com/ggerganov/llama.cpp

APPENDIX

A. Domain-Specific Language Example

```
constexpr auto digit =
  NTerm(cs<"digit">());
constexpr auto d_digit = Define(digit,
Repeat(Alter(Term(cs<"1">()),
  Term(cs<"2">()), Term(cs<"3">()),
  Term(cs<"4">()), Term(cs<"5">()),
  Term(cs<"6">()), Term(cs<"7">()),
  Term(cs<"8">()), Term(cs<"9">()),
  Term(cs<"0">()))));

constexpr auto number =
  NTerm(cs<"number">());
constexpr auto d_number = Define(number,
  Repeat(digit));
constexpr auto add = NTerm(cs<"add">());
constexpr auto sub = NTerm(cs<"sub">());
constexpr auto mul = NTerm(cs<"mul">());
constexpr auto div = NTerm(cs<"div">());
constexpr auto op = NTerm(cs<"op">());
constexpr auto arithmetic =
  NTerm(cs<"arithmetic">());
constexpr auto group =
NTerm(cs<"group">());

// There is no operator
precedence calculation
constexpr auto d_add = Define(add,
  Concat(op, Term(cs<"+">()), op));
constexpr auto d_sub = Define(sub,
  Concat(op, Term(cs<"-">()), op));
constexpr auto d_mul = Define(mul,
  Concat(op, Term(cs<"*">()), op));
constexpr auto d_div = Define(div,
  Concat(op, Term(cs<"/">()), op));

constexpr auto d_group = Define(group,
  Concat(Term(cs<"(">()), op,
  Term(cs<")">())));
constexpr auto d_arithmetic =
  Define(arithmetic,
  Alter(add, sub, mul, div));
constexpr auto d_op = Define(op,
  Alter(number, arithmetic, group));

constexpr auto ruleset = RulesDef(d_digit,
  d_number, d_add, d_sub, d_mul, d_div,
  d_arithmetic, d_op, d_group);
```