

SuperCFG parser generator

Remi Savchuk¹, Supervised by Ekaterina Trofimova¹

¹National Research University Higher School of Economics

Parser generators



Parser generators

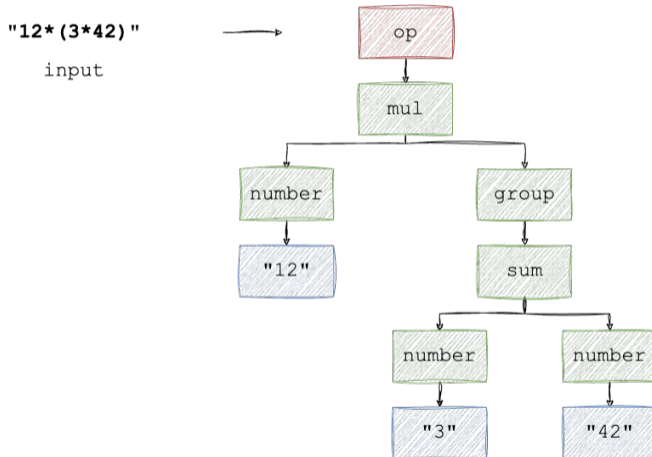


Figure: Example of a parser generator output

Types of parser generators

- ▶ Recursive-descent parser generators are the simplest type of parsers, which recursively walk over a tree
- ▶ This type of parsing cannot handle recursive grammars
- ▶ Shift-reduce parser generators build the result from the bottom instead
- ▶ Donald Knuth designed the LR parser, which treats the grammar as an automata

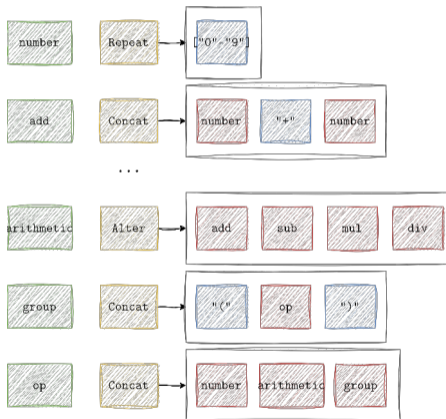


Figure: Example of a parser generator grammar

Problems

- ▶ LR parsers can only work with deterministic states, such that there is only one possible reduction
- ▶ Algorithms like GLR and Earley's parser explore the whole set of possible reductions
- ▶ Such algorithms can be called static. For ambiguous grammars, another approach can be used

Dynamic parser generator



Parser structure

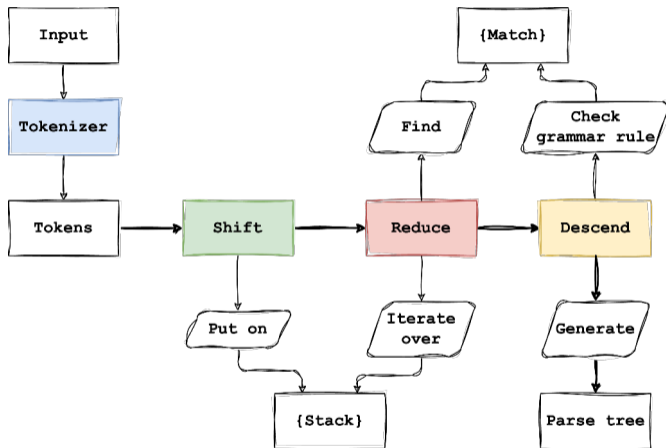


Figure: Dynamic shift-reduce algorithm pipeline

Reduce routine

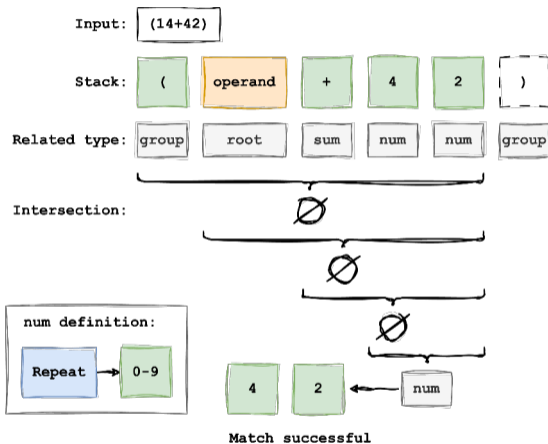


Figure: Example of a reduce routine at some stack state for a calculator grammar

Heuristic ambiguities resolver

- ▶ The base algorithm chooses too many possible candidates
- ▶ It can be expanded to analyze context and make decision whether to reject the match candidate
- ▶ This allows the parser algorithm to be expanded to parse larger sets of grammars
- ▶ Reducibility checker $\mathcal{RC}(1)$ is a module that checks if a match candidate can be reduced one step ahead in the future
- ▶ Prefix-based context analyzer uses prefixes and postfixes of a rule in order to analyze current context

Grammar serialization

Baking function \mathcal{B} performs compile-time serialization of grammars into a custom EBNF-like textual representation

This mechanism is essential for integrating symbolic grammar definitions with external components such as language models

It can also be used for an iteration over a superset of grammars

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

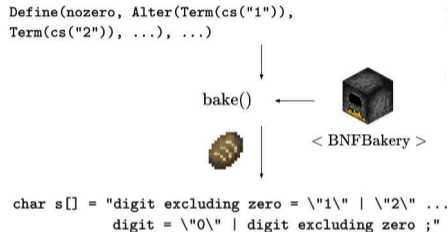


Figure: Compile-time serialization process

Complexity analysis

- ▶ As the parser algorithm is dynamic, it performs much slower than the linear-time static shift-reduce parsers
- ▶ The shift routine is linear in $\mathcal{O}(n)$, while reduce routine needs to perform an $\mathcal{O}(|\mathcal{S}|^2)$ iteration over the stack \mathcal{S} : $\mathcal{O}(n \cdot |\mathcal{S}|^2 \cdot |G|) = \mathcal{O}(n^3)$
- ▶ 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
- ▶ Non-deterministic static parsers like CYK and Euler also operate in $\mathcal{O}(n^3)$

Debugger

The debugger interface displays several panels for parsing and grammar analysis:

- SR PARSER ROUTINE**: Shows the current state of the shift-reduce parser. The stack contains '(', 'op', ',', and 'op'. The 'rel' field is empty. The 'gro' field is 'DESCEND'. The 'arr' field is empty. The 'CNO' field is '1/2'. The 'group' field is '["(", "op", ",", "op"]'. The 'op' field is 'op'.
- ABS SYNTAX TREE**: Shows the abstract syntax tree. The root node is 'op', which has two children: 'string' and 'char'. The 'string' node has a child 'string' with value 'abc'. The 'char' node has a child 'string' with value 'asdf'.
- EBNF Grammar**: Shows the grammar rules for the language. The rules are: char := [{"a"-"z"}]*, string := (char)*, op := (string | group | array), group := ("(" + op + (("(", "op")*) + ")"), array := ("[" + op + (("(", "op")*) + "]").
- REVERSE RULES TREE**: Shows the reverse rules tree. The rules are: char -> string, string -> op, op -> group array, group -> op, array -> op.
- FIX Heuristic**: Shows the heuristic rules. The rules are: char -> - : -, string -> - : -, op -> - : -, group -> ("op") : ("op"), array -> ["op"] : ["op"].

At the bottom of the interface, there is a status bar with the following text: TAB Next Win, O Open Win, Q Quit, M Move Win, E Destroy Win, T Theme, A About.

The project



Features

- ▶ The project uses C++20 metaprogramming features for building compile-time parser generator structures
- ▶ Grammar initialization is done in compile-time, and the parser generator is used as a header-only library
- ▶ Almost all of the existing parsers generate an intermediate source file, which must be compiled
- ▶ The only language that has a more powerful metaprogramming system than C++ is Haskell. BNFC-Meta is another embedded parser generator (*AFAIK there are no more shift-reduce embedded parsers in existence*)

C++ hacks

- ▶ Type morphing casts each type $T\langle t_1, t_2, \dots, t_n \rangle$ into $T\langle \lambda(t_1), \lambda(t_2), \dots, \lambda(t_n) \rangle$
- ▶ Can be used as a constexpr type constructor (`using ...
decltype(type_morph_t<...>())`), which allows us to declare class members without the use of `auto`. Still causes `abort()` in g++ 15.2.1 for some reason (*usage: parser structures*)
- ▶ C++20 variadics support allows to apply set operations to tuples (*usage: parser structures*)
- ▶ Dynamic hashtable stores each element as a `std::variant`, which allows to dynamically store element types and access them using `std::visit` (*usage: parser stack*)
- ▶ We can even cast a tuple into a homogeneous type and return it down the stack, but the compilation time may be longer than the heat death of the universe

Next steps

- ▶ Project is still in development, context analyzer is still in WIP
- ▶ We need to formally define the parser properties
- ▶ This parser generator is going to be used in a decoding stage of an LLM pipeline. In particular, the grammar iteration will be used for finding an optimal grammar for a particular LLM



Figure:
<https://github.com/enaix/SuperCFG>