A Formal Description of an Algorithm Suitable for Parsing the Language of Mathematics

CICM 2025, Brasília, Brasil

Luka Vrečar, Joe Wells, Fairouz Kamareddine

Heriot-Watt University

October 10, 2025

Motivation

- The language of mathematics is ambiguous and hard to parse
- Our previous work deals with parsing mathematical documents
  - Existing GLR-based libraries proved inadequate

Motivation

- The language of mathematics is ambiguous and hard to parse
- Our previous work deals with parsing mathematical documents
  - Existing GLR-based libraries proved inadequate

We tried non-GLR Python parsing libraries and `parglare`, which is a Python GLR implementation. Even `parglare` proved inadequate for certain grammars.

DynGenPar - a potential solution

Stands for Dynamic Generalized Parsing

- The grammar can be updated during parsing - dynamic
- Parses exhaustively, with any CFG - generalized

Designed to parse the language of mathematics

- Part of the FMathL project at the University of Vienna
- Presented at CICM in 2011 and 2012 (Kofler and Neumaier)

We distinguish two parts of DynGenPar

Abstract definition

A non-deterministic algorithm

- finds one parse tree for an input sentence
- there is a sequence of non-deterministic choices that lead to a parse tree
- not clear how to implement the described algorithm well

C++ implementation

Deterministic and finds (supposedly) all parse trees for a given input sentence

- concurrency mechanism synchronizing on input tokens ("continuations")
- additional features like parse tree "compression", support for PMCFGs, next token constraints, etc.
- relies on outdated versions of external libraries (Qt)
- under-documented and hard to follow

C++ implementation

Deterministic and finds (supposedly) all parse trees for a given input sentence

- concurrency mechanism synchronizing on input tokens ("continuations")
- additional features like parse tree "compression", support for PMCFGs, next token constraints, etc.
- relies on outdated versions of external libraries (Qt)
- under-documented and hard to follow

The algorithm relied on Qt's implementation of data structures like lists, hash maps, etc.

1. Continuations represent parsing threads, which suspend when they need more symbols from the input
2. Once a new symbol is consumed, the thread resumes
3. We call them continuations because they literally continue computation where they suspended
4. They allow us to find all parse trees while only going through the input once

In this paper

- Created a new, more rigorous definition based on a simplified version of the implementation
  - Since our CICM submission, we simplified it further
  - This is the version we'll present today
  - A new paper will be available soon
- Provided an implementation that matches the definition as closely as possible
- Proved some properties of the newly-defined algorithm

Running example

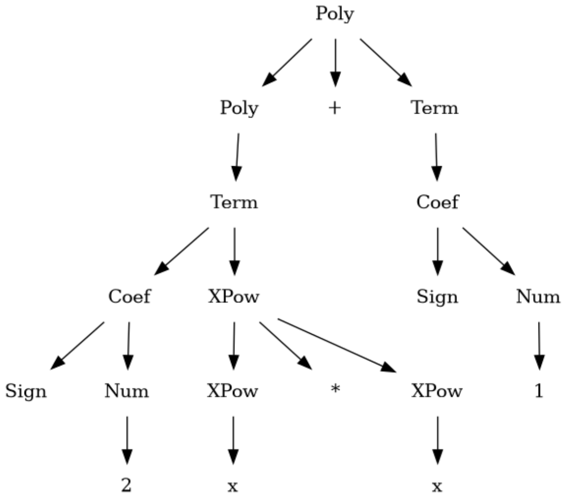
- Simple grammar for polynomials

$$
\begin{aligned}
\textbf{Poly} &\rightarrow \textbf{Poly} + \textbf{Term} \mid \textbf{Poly - Term} \mid \textbf{Term} \\
\textbf{Term} &\rightarrow \textbf{Coef XPow} \mid \textbf{Coef} \mid \textbf{XPow} \\
\textbf{Coef} &\rightarrow \textbf{Sign Num} \\
\textbf{Sign} &\rightarrow \text{-} \mid \varepsilon \ (\varepsilon \text{ represents the empty string}) \\
\textbf{XPow} &\rightarrow \text{x} \mid \textbf{XPow * XPow} \\
\textbf{Num} &\rightarrow 1 \mid 2
\end{aligned}
$$

- Input sentence $2x * x + 1$
- **Poly** is our start symbol

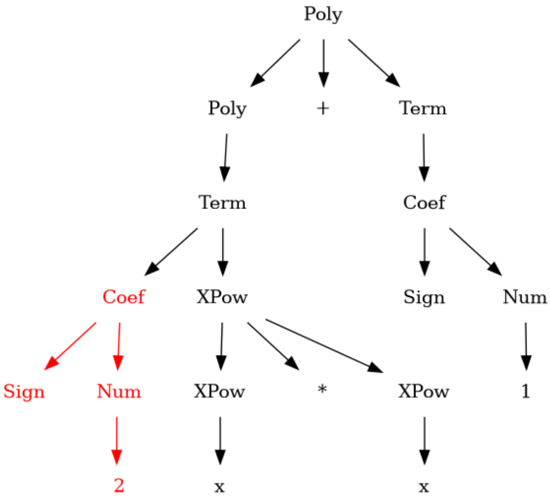
Running example

- $2x * x + 1$ is unambiguous
- Only (parse) tree - see below
  - In this talk: an *extremely* abridged version of finding it



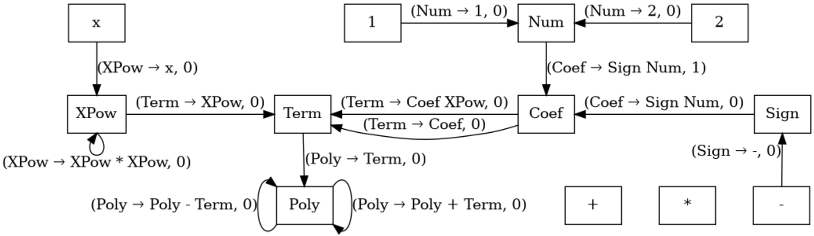
Nullable nonterminals

- Nullable nonterminals can parse the empty string (ε)
- For our grammar, we have exactly one - **Sign**



Initial graph

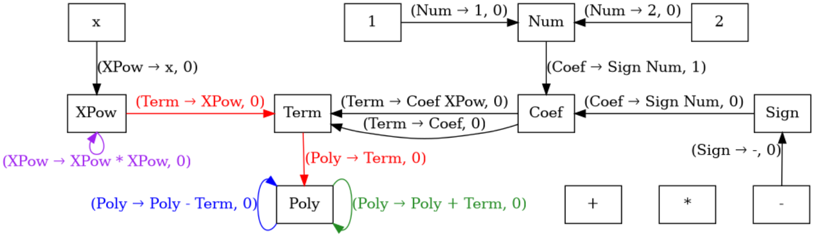
- Symbols are nodes, edges are directed, labelled with (rule, n)
- Edge from, e.g., **Num** and **Coef** means we can use a tree with root **Num**, to build a tree with root **Coef**
  - Must find parse trees for all symbols on RHS of the rule
  - n tells us how many nullable nonterminals need trees that parse ε



Neighborhoods

We explore the initial graph using Kofler's notion of neighborhood

- There can be infinitely many paths between two symbols (e.g., **XPow**, **Poly**)
  - Considering complete paths is unfeasible
- Take one step along all outgoing edges, check if we can reach **Poly**
- Once we complete a parse tree for the first rule, consider the new neighborhood

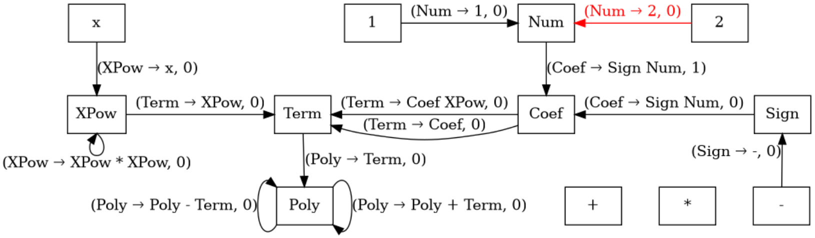


Running example - initial setup

- Start with continuation $\mathcal{C} =$ **done Poly**
- Main loop
  1. Consume a symbol from the input, process each continuation with it
  2. This gives us new continuations and parse trees
  3. Return trees if no more input, otherwise discard them
- In the end, we have all the parse trees for the input sentence with root **Poly**
  - For our example, this would be just 1 tree

neighborhood(2, **Poly**)

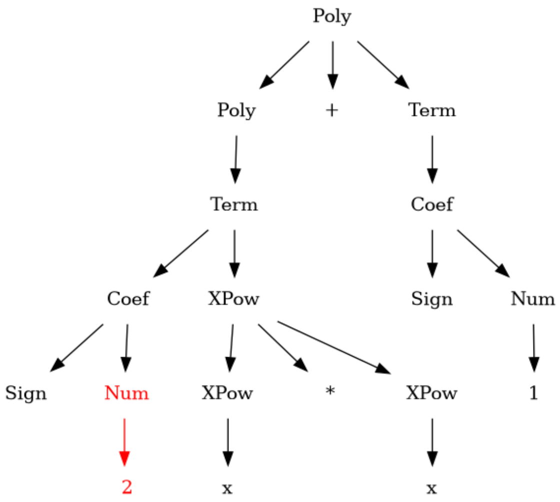
2 is the first symbol in the input, we want to build a tree with root **Poly**



Only one rule - **Num** $\rightarrow$ 2

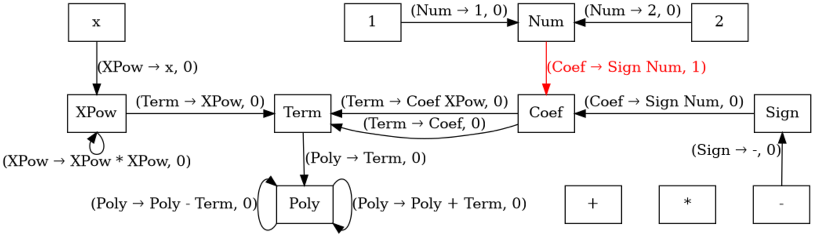
Processing 2, continued

Immediately build subtree for the rule **Num** $\rightarrow$ 2



The subtree doesn't have root **Poly**, so turn to the initial graph again

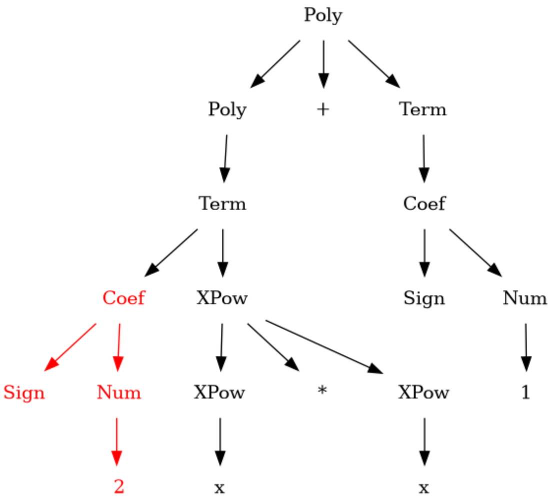
neighborhood(**Num**, **Poly**)



- Only one rule - **Coef** $\rightarrow$ **Sign Num**
- The 1 indicates we need a parse tree for the empty string with root **Sign** (possible via the rule **Sign** $\rightarrow$ ε)

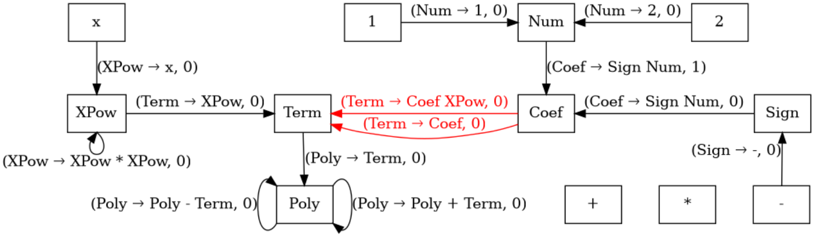
Running example - starting off

We get the following subtree



We haven't reached **Poly**, so consider *neighborhood*(**Coef**, **Poly**).

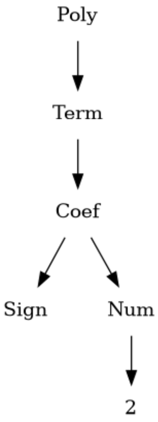
neighborhood(**Coef**, **Poly**)



- Two rules: **Term** $\rightarrow$ **Coef** and **Term** $\rightarrow$ **Coef XPow**
- Computation splits to consider them both

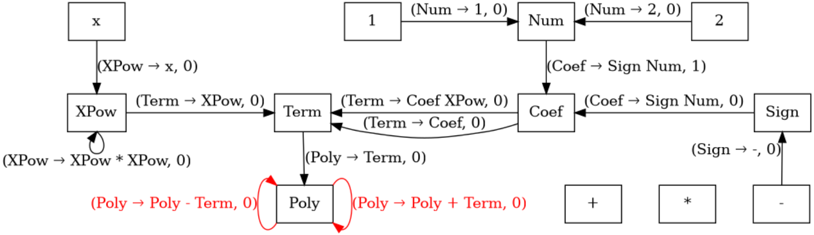
Processing 2 - considering **Term** $\rightarrow$ **Coef**

We get the following subtree via **Term** $\rightarrow$ **Coef** and **Poly** $\rightarrow$ **Term** (it gets returned)



To account for left-recursive rules for **Poly**, consider *neighborhood*(**Poly**, **Poly**)

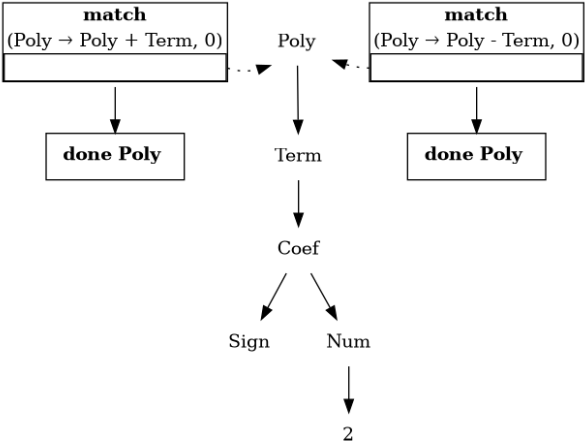
neighborhood(**Poly**, **Poly**)



Both rules require additional symbols, so parsing is suspended and new continuations are created

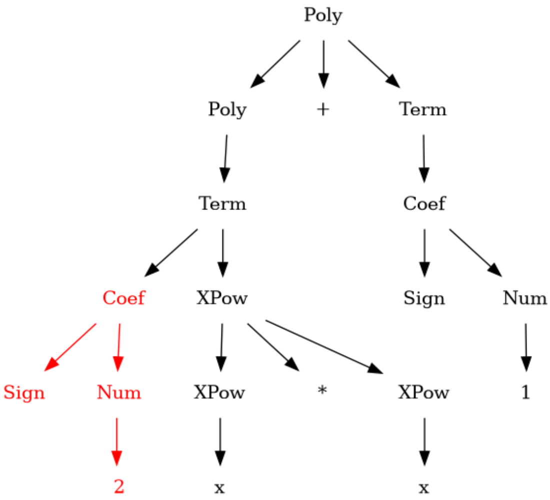
Processing 2 - considering **Term** $\rightarrow$ **Coef**

The continuations, visualized



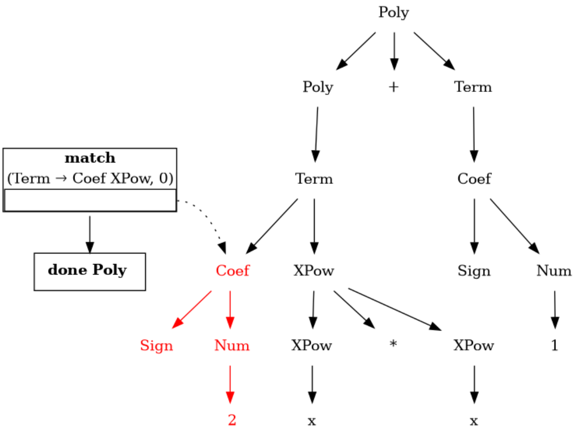
Processing 2 - considering rule **Term** $\rightarrow$ **Coef XPow**

Recall the subtree with root **Coef** that has already been built



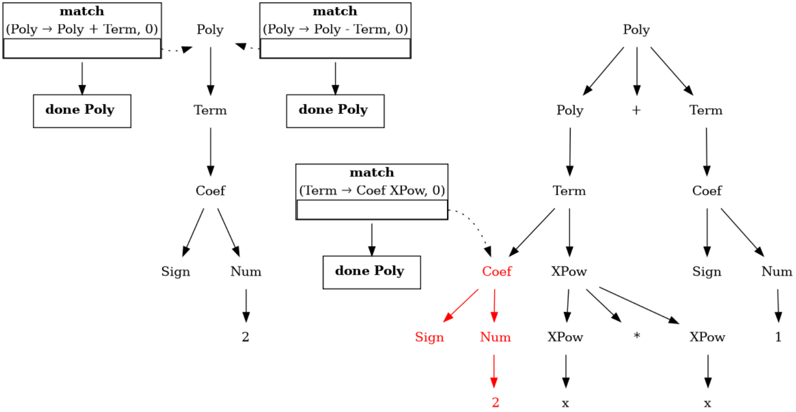
Processing 2 - considering rule **Term** $\rightarrow$ **Coef XPow**

Additional symbols are needed, so create a continuation

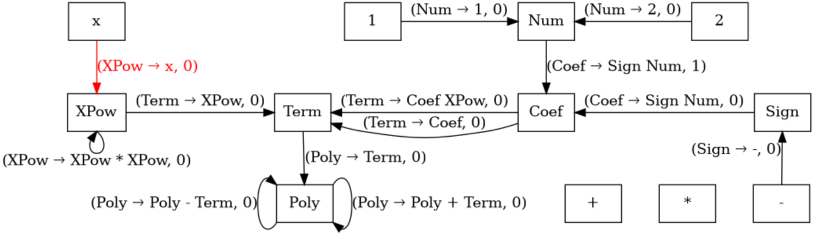


Processing 2 completed

After processing 2, we have the following



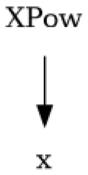
neighborhood(*x*, **XPow**)



There is one rule - **XPow** $\rightarrow x$

Processing x, continued

We have a subtree with root **XPow**

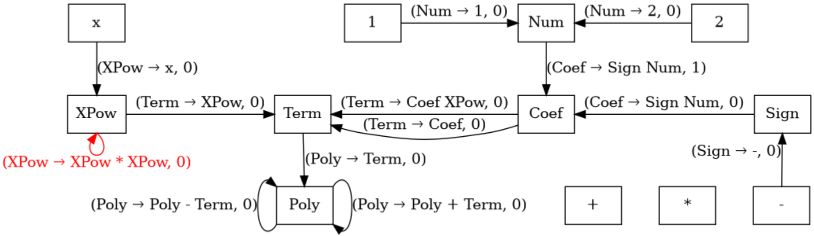


Two things to do with this subtree

- Complete a tree for the rule **Term** $\rightarrow$ **Coef XPow**
  - This will fail, like the subtree we built when processing 2
- Consider *neighborhood*(**XPow**, **XPow**) to account for left recursion

Processing x - considering left recursion at **XPow**

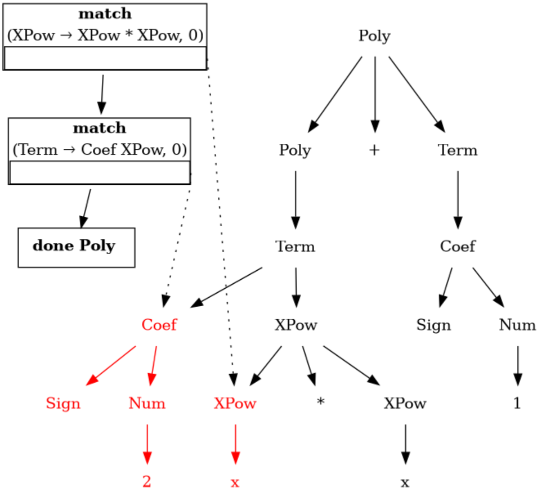
Consider *neighborhood*(**XPow**, **XPow**)



We get one rule - **XPow** $\rightarrow$ **XPow** $*$ **XPow**

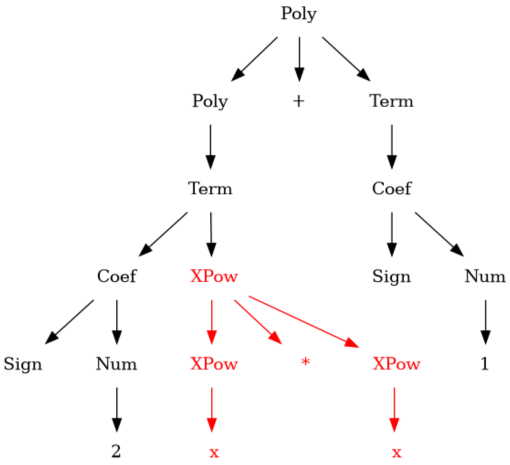
Processing x - considering left recursion at **XPow**

More symbols needed, create a new continuation



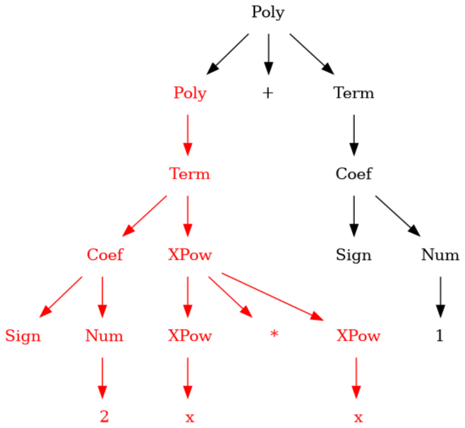
Completing the subtree for the rule **XPow** $\rightarrow$ **XPow** $*$ **XPow**

Consuming more symbols, attach leaf tree for * and another tree with root **XPow** to complete the tree



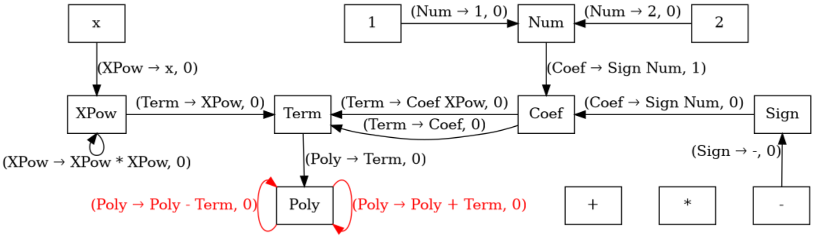
Processing the second x - building a tree for **Poly**

We build the following subtree



Account for left recursion at **Poly**, so consider *neighborhood*(**Poly**, **Poly**)

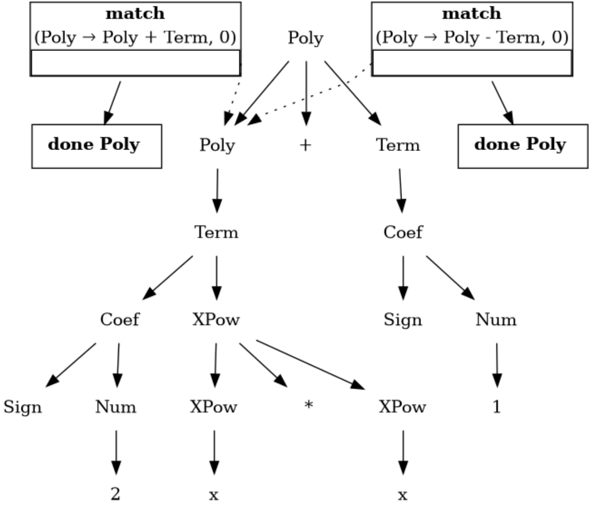
neighborhood(**Poly**, **Poly**) - a reminder



We have two rules for which we create continuations

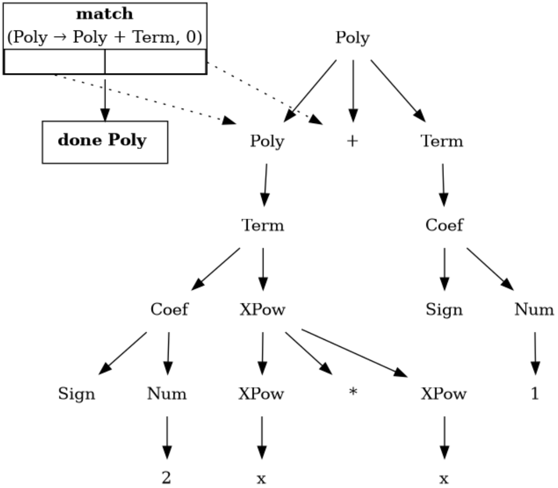
Processing the second x - finishing up

We have two continuations, the right one will get discarded



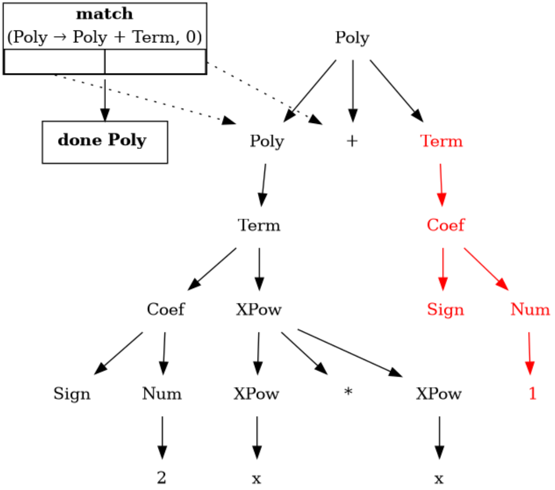
Processing +

Consume + from the input, attach its leaf tree to the continuation



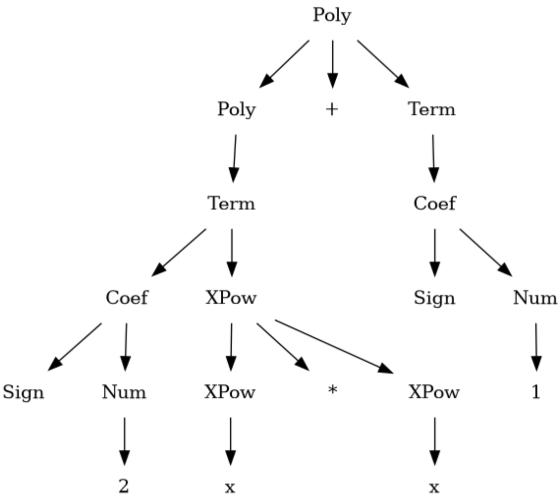
Completing the parse tree

We consume 1 from the input, and build a subtree with root **Term** (via **Num** $\rightarrow$ 1, **Coef** $\rightarrow$ **Sign Num**, and **Term** $\rightarrow$ **Coef**)



The full parse tree for $2x * x + 1$

We have completed the parse tree!



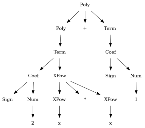
2025-10-10

A Formal Description of an Algorithm Suitable for Parsing the Language of Mathematics

└The full parse tree for $2x * x + 1$

The full parse tree for $2x * x + 1$

We have completed the parse tree!



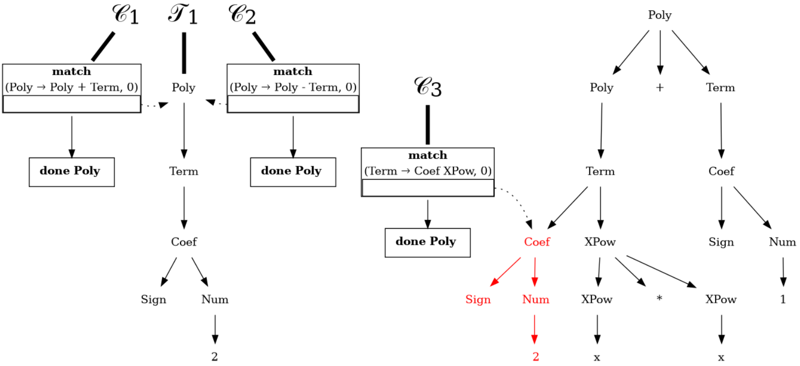
So is this!

Why we know the algorithm works

- Let $\mathcal{C}$ be a continuation that has been createad by processing k input symbols
- Let T_k be the set of all trees described by $\mathcal{C}$
- Processing $\mathcal{C}$ with the $k+1$-th symbol gives us new continuations and trees
- Continuations and trees partition T_k
- For every completed parse tree there is a unique sequence of continuations that produces it

A partition example

Recall the tree and continuations we obtained by processing 2



A partition example

Set of all parse trees with root **Poly**

Image is not to scale.

A partition example

Set of all parse trees with root **Poly**

Parse trees whose leftmost terminal leaf is 2
(obtained by processing **done Poly** with 2)

Image is not to scale.

A partition example

Set of all parse trees with root **Poly**

Parse trees whose leftmost terminal leaf is 2
(obtained by processing **done Poly** with 2)

$\mathcal{T}_1$ | $\mathcal{C}_1$ | $\mathcal{C}_2$ | $\mathcal{C}_3$

Image is not to scale.

A new implementation

- Needs only the Python standard library
- Optional detailed logging capabilities
  - helped us develop the proofs
- Matches the definition as closely as possible
  - Some changes were made for efficiency
- Freely available with some examples and utilities
  - `https://github.com/LVrecar/dyngenpar`

Future work

- Formalization of the algorithm with a proof assistant
- Add parse tree/continuation compression (for improved efficiency with ambiguous input)
- Explore the time and space complexity of the algorithm
- Other extensions included in Kofler's original implementations should be considered (e.g., PMCFGs)