Natural Language Processing (2025), 1–7
doi:10.1017/xxxxx

ARTICLE

The Third Way: Generalized Operator Precedence Parsers

Greg McClement

greg.mcclement@protonmail.com

(Received xx xxx xxx; revised xx xxx xxx; accepted xx xxx xxx)

Abstract
The papers presents ideas that can be used to generalize an operator precedence parser so that natural languages can be parsed thus creating a third way in addition to grammar and neural net based approaches.

1. Introduction

Machine-based natural language processing started development in the 1950's after the publication of Chomsky's Syntatic Structures. Following that, the main approach to processing natural language utterances was based on programmed solutions using grammars. That approach has been recently superceeded by neural net based solutions whose current iteration is LLMs. LLM's have greated expanded the capability of NLP systems but the cost was loss of control over the behaviour of the systems. This has manifested in what is described as 'halucinations', the problem of overriding neural net configuration to jail break functionality, the need for hand labelled data and the problem of dimishing returns from scaling data and hardware. This paper describes a brief overview on a third approach to natural lange processing, a generalization of operator precedence parsers called a generalized operator precedence parser (GOPP).

An operator precedence parser is a simple shift-reduce parser that can of parse a subset of LR(1) grammars. This is the starting point. The generalizations presented bring capabilities into the context sensitive languages. The generalizations are intuitive and effective. They allow handling of structures that the grammar based approaches have difficulty encompassing. Since this approach is a non-sequeter from has been happening what I will be presenting is a general overview. There is a detailed implementation including practical POS's at http://theprogrammablemind.com that is based on ideas outlined here. I regard this implementation not a standard but as just a way to implement the generalizations being presented.

2. The Basics

In 1992 in a lab at UBC, I was eating my sandwich waiting for the Prolog lab to start and a sentence came into my head, "the cat went to the store" while I was staring at some Prolog code in VI. Prolog uses an operator precedence grammar and so has the ability to define operators. When I looked at the sentence I saw "the" was an prefix operator, "cat" and "store" were constants, "went" was an infix operator, "to" was a prefix operator. The order of precendence was "the", "to", "went". This can be coded up in Prolog as

```
:- initialization(main).
:- op(300, fy, the).
```

© Cambridge University Press 2025

```
:- op(400, fy, to).
:- op(500, xfy, went).

main :- write_canonical(the cat went to the store).
```

the output is

```
went(the(cat),to(the(store)))
```

As mentioned early operator precedence parsers can only parse a subset of LR(1) grammars so some additions need to be made in order to encompass natural languages.

3. Contexts (Property Bags as Operators)

The typical use for operator precendence parsers is arithmetic expressions, such as 2+4/6. In that case the constants are numbers. For the Prolog example the constants are atoms. Based on pratical experience I ended up using property bags as operators. That allows a multi dimentional collection of information to be passed through the calcuation. So far that has been sufficient for all the computations. I will call these property bags, contexts.

4. Dead Operators

For GOPP the output of applying the parser is a sequence of contexts not necessarily a single context. In the implementation mentioned earlier, there is a useful semantics that can process a sequence of contexts as the output. While the parser is running if an operator cannot apply it becomes a dead operator. That is an operator that is no longer considered for application. This will be referred to as a constant and has the same role. I was calling the process of mapping from the utterance to a data structure, the interpretation and the processing of the data structure that follows, the semantics.

5. Bridges

For the expression 2+3 the result of applying the + operator is 5 because arithmetic addition was performed. For GOPP, that is not sufficient since contexts have been chosen as the operator values. When an operator is applied the input values would be the arguments and the operator as it appears in the current calculation. The function that calculates the output of the expression is called a bridge since it moves data from the input contexts to the output context. In the aforementioned implementation this manifests as the property 'before', which is the arguments to the left of the operator, 'after' which are the arguments to the right of the operator and 'operator' which is the context of the operator. In the implementation I used javascript notation for calculating the context which I will use in the rest of this document. An example of a bridge for addition is

```
{ ...next(operator), sum: before[0] + after[1] }
```

6. Selectors

In OPP for arithmetic expression all the arguments are numbers so there is no need to worry about if the argument makes sense for the operator. In GOPP, the arguments are contexts that may or may not be what the operator accepts. As seen later there are some operators that apply not based on priority but based on possibility. In order to account for this, the operator have selectors for the arguments. The selectors look at the contexts to the left and right of the operator and select the argument that the operator accepts because on criteria. If the arguments are not selected the application of the operator will fail and the operator becomes dead. Operators are identifed by the operator name and level. This will be written as <name>/<level>. Where name is the name of the operator and level is an integer. The next sections explains the level.

For example from the implemention, the following defines an operator called 'addition' with level 0 that takes two arguments of type number with any level

```
((number/*) [addition|+] (number/*))
```

Arguments are wrapped with (). Operator are wrapped with [] or <>. For operators wrapped in [] the result of applying the expression is assumed to be <operator>/<level>+1 . For operators wrapped in <> (passthrough operators) the result of applying the expression will be one of the arguments. Here is a more complex example

```
((person/*) [buy] (<determiner> (car/*)))
```

This defines buy/0 that has a selector on the left for contexts of type person/* and a context on the right of type car/* because the passthrough marker was used. It also defines determiner/0 that has a right selector for contexts of type car/*. For the selectors I will use this notation

```
(leftArg1, leftArg2...leftArgk)operator/level(rightArg1, rightArg2...rightArgl)
```

The implemention has a more detailed expression language for selections but for this overview the types is sufficient.

7. Hierarchy

The hierarchy system is based on the operator name not the operator/level . It makes sense that it would be operator/level then one could say operator/0 isA word and operator/1 is a concept. In practice, when I tried implementing that the code became very unreadable and configuring it correctly was challenging. Perhaps I need more research on the to find a good notation or implementation but currently I have left the hierarchy being based on the operator name then argument selectors can be triggered on the level. That was the more readable solution and allows argument selection that distinguished words from concepts.

8. Operators that evaluate to other operators

For OPP, evaluating an operator such as '+' in "2+2" results in a constant. For a GOPP, the result is another operator. The notation is that each operator has a level which is an integer. The level starts with zero and each application can increase the level. Then for each level there is a corresponding selector and bridge function. Let's look at an example, 'and'. The notation will be <operator>/<level> . In Prolog, and/2 would mean the and operator has arity 2. For this case the arguments have type so I dont use that notation. Instead and/2 would mean the second level of the operator. Operator levels start at zero. And can be defined as seen in table 1.

For the parse, assume "cats", "dogs" and "goats" are "andAble". For the implementation every context has a marker property whose value is the operator of the context. Below I will shorten { marker: cat/0 ... } to just cats/0 etc...

Table 1. Definition of the 'and' operator

Operator	Bridge
(andAble)and/0(andAble)	{ ...next(operator), list: append(before, after) }
(andAble)and/1	{ ...operator, list: append(before, operator.list) }

next(operator) is a context where the next level of the operator is used as the marker.

Table 2. Applying the bridges of the 'and' operator

Contexts	Before	After	Operator
[cats/0, dogs/0, and/0, goats/0]	[dogs/0]	[goats/0]	and/0
[cats/0, { marker: and/1, list: [dogs/0, goats/0] }]	[cats/0]	[]	and/1
[marker: and/1, list: [cats/0, dogs/0, goats/0]]			

for brevity, cats/0 is understood to be { marker: cats/0 } etc...

Operator levels support this kind of calculation where, after the application, the operator definition changes.

9. Zero-Arity Operators

Adding zero arity operators allows the creation of a type system. Instead of regarding 'cat' and 'store' as constants they can be thought of as zero arity operators. Then there are no constants there are just operators. Then we can generalize the operators definition to include the type of the arguments along side a hierarchy system. For example instead of "went" being xfy, "went" can be (moveable)fy. "cat" and "moveable" are zero arity operator where "cat" is a type of "moveable". Then the parser can do type checks for the arguments when applying the selectors. For the simple example, that is not really necessary but for complicated sentence the type is important in resolve the interpretation.

10. Law of Distribution over Lists

Having a "law of distribution" turns out to be handy. Consider the utterance "a cat and dog". Assume a/0(concept) defined in Table 3 and that dog and cat are concepts

Table 3. Definition of the 'a' operator

Operator	Bridge
a/0(concept)	{ ...after[0], det: 'a' }

this is a passthrough operator where the context that results is not a higher level of the operator applied

The parse would start out like as shown in Table 4.

Now we are stuck because and/1 is not a concept. Even if it was we dont want to get

```
{ marker: and/1, det: 'a', list: [dog/0, cat/0] }
```

we want the a/0 applied to dog/0 and cat/0. This can be achieved with a distribution of a/0 over the element in list. The appropriate type checks apply to all the elements in list. If that works then a/0 is applied to each element and repackaged into the list like

Table 4. Applying the bridge of the 'and' operator first

Contexts	Before	After	Operator
[a/0, dog/0, and/0, cat/0]	[dog/0]	[cat/0]	and/0
[a/0, { marker: and/1, list: [dogs/0, cat/0] }]			

assume and/0 applies first

```
{ marker: and/1, list: [{ dog/0, det: 'a'}, { cat/0, det: 'a' }] }
```

The result is nice and tidy. I have not had time to explore if other algebraic operations have a use.

11. Variable Priority Operators

Instead of having a fixed priority for operators some operators need a variable priority. The operator is applied based on a condition being present rather than being the highest priority operator available. For example consider the utterances "a dog and a cat" and "a dog and cat". For the former and/0 should apply after a/0 and for the latter and/0 should apply before. Clearly a fixed priority number is not getting the job done. For this example if a/0 was not andAble/0 and dog and cat are andAble then "a dog and a cat" parse can be seen in Table 5 and "a dog and cat" parse can be seen in Table 6. The and/0 operators applied as soon as possible rather than based on a fixed priority.

11.1 Example 1 - "a dog and a cat" where 'and' applies before 'a'

Table 5. Applying the 'a' operator first

Contexts	Before	After	Operator	Note
[a/0, dog/0, and/0, a/0, cat/0]		[dog/0]	a/0	and/0 does not apply since
				'a' is not 'andAble'
[{dog/0, det: 'a'}, and/0, a/0, cat/0]		[cat/0]	a/0	apply second a/0
[{dog/0, det: 'a'}, and/0, {cat/0, det: 'a'}]	{dog/0, det: 'a'}	{cat/0, det: 'a'}	and/0	both args are 'andAble'
[{ and/1, list: [{dog/0, det: 'a'}, {cat/0, det: 'a'}]}				

11.2 Example 2 - "a dog and cat" where 'a' applies before 'and'

Table 6. Applying the 'a' operator first

Contexts	Before	After	Operator	Note
[a/0, dog/0, and/0, cat/0]	[dog/0]	[cat/0]	and/0	and/0 does apply
[a/0, { and/1, list: [dog/0, cat/0]}		{ and/1 ...}	a/0	use distribution
[{ and/1, list: [{dog/0, det: 'a'}, {cat/0, det: 'a'}]}				

12. Implicit/Convolution Operators

I had trouble deciding whether or not to call this implicit or convolution operator. I went with convolution but implicit is still a great name. Maybe I will switch back to that. Who knows. The future is a mystery yet to unfold. A convolution operator is an operator that does not appear in the utterence directly but is applied because its arguments appear in the utterance. Originally when I implemented counts I made it a prefix operator over a countable as seen in Table 7.

Table 7. Definition of the 'count' operator

Operator	Bridge
count/0(countable)	{ ...after[0], count: operator }

where count is an integer type. It didn't feel right because I was making a number a prefix operator in some cases and in other cases (for example 2+2) not a prefix operator (the system I implemented can handle that kind of ambiguity). To solve this and a whole bunch of other types of phrases the convolution operator is used.

Table 8. Definition of the 'a' operator

Operator	Bridge
(number/*)counting/0(countable/*)	{ ...after[0], count: before[0] }

Where the counting operator is marked as a convolution. I called it convolution because the computation works like a convolution. Let's see an example in Table 9.

Table 9. Applying the counting/0 operator first

Contexts	Before	After	Operator	Note
[20/0, goats/0]	[20/0]	[goats/0]	countable/0	arguments match
[{ goats/0, count: 20/0}				

Since the operator is virtually present the prioritiation rules still apply.

13. Manifestation of Grammar Concepts

Concepts from the grammar theory of languages manifest in GOPP. Concepts such as verbs, prepositions, determiners etc... manifest as operator priorities. Determiners apply before prepositions which apply before verbs. Determiners are prefix operators as are prepositions. Verbs can be infix, prefix or postfix depending on the verb. So there is an interesting relationship between grammars and GOPP where the later can be seen as building blocks for the former similar to similar to a proton being composed of two up quarks and one down quark.

14. Summary and Conclusion

The implementation (http://theprogrammablemind.com) of these concepts shows that this approach is effective in producing useful systems that can handle natural language input in real world applications. Seeing the effectiveness of this approach with other languages would be interesting. I have a bit of code that does french and matches the genders but there is nothing extensive. This is a new approach so would benefits from ideas from other people and applications.

References

Chomsky, N. (1957). *Syntactic Structures*. The Hague: Mouton

Pratt, V. (1973). Top Down Operator Precedence., *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 41–51. Boston.