École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Top Down and Bottom Up Approaches to Parsing

Author:

Aarrya Saraf, École Polytechnique

Advisors:

Dr. Noam Zeilberger, Laboratoire d'informatique de l'École polytechnique
Dr. Dale Miller, Laboratoire d'informatique de l'École polytechnique

Academic year 2023/2024

Abstract

Parsing algorithms form an important component of technology such as compilers and interpreters, as well as in computational linguistics. There is an old connection between parsing and deduction under which parsing a string may be compared with building a proof in a logic. This connection has been used for example to express parsing algorithms using logic programming before. A natural question is how the dichotomy between top-down and bottom-up approaches to parsing may be reflected logically. This may be approached from a proof-theoretic perspective, using advances in the understanding of forward and backward chaining in proof construction. This paper focuses on fixing Context-Free Grammars, and then finding and analyzing the different top-down and bottom-up approaches to parse words following it.

Contents

1 Introduction and Related Work **4**

2 Simple Calculator **4**
2.1 Tokenizing . . . 5
2.2 Evaluation . . . 6

3 Top Down vs Bottom Up algorithms **8**

4 Parsing Algorithms and Grammars **9**
4.1 Stack Based Parsing . . . 10
4.1.1 Look Ahead and Dynamic Programming . . . 11
4.2 CKY Parsing . . . 12

5 Correctness **15**
5.1 Soundness and Completeness . . . 15
5.1.1 Automatic Parse Tree . . . 16
5.2 Guaranteed Termination . . . 18

6 Future Work **19**
6.1 Earley Algorithm . . . 19
6.2 Better Parsing Algorithms . . . 21
6.3 Term Sharing and Common SubFormula Elimination . . . 21

7 Conclusions **24**

8 References **25**

A Appendix **27**
A.1 Other grammars used . . . 27
A.1.1 Even number of 1s in a binary string . . . 27
A.1.2 Binary Strings ending in 0011 . . . 27
A.2 CYK Algorithm . . . 28
A.2.1 Auxillary functions for . . . 28
A.2.2 Full grammar for the Prolog example . . . 28
A.3 Simple calculator CNF Prolog grammar . . . 29
A.3.1 Grammar in CNF . . . 29
A.3.2 Auxilliary functions . . . 30
A.4 Auxilliary functions in Automatic Parse Tree generator . . . 31
A.5 Auxilliary functions in Future Work . . . 31

1 Introduction and Related Work

Given a fixed formal grammar, parsing is the process of analyzing a string to determine whether or not it follows the grammar [27]. This is generally preceded by converting the string into a list of its constituent tokens as seen in the grammar. The grammars are normally written such that there is a maximum of one way to parse a string and structures known as parse trees display this way. Parsing is an integral part of compilers. In fact, they are so important that they are not made manually. Instead, some restricted grammar is fed into specific programs which themselves spit out the code for a parser. The most famous example of this would be Yet Another Compiler Compiler (YACC) [10].

There is an old connection presented by Pereira and Warren [16] which presents the basis for the connection between parsing a string and building a proof in a logic related to the grammar. Since then a lot of work has been done on it. A central paper to our approach has been by Shieber, Schiebes, and Pereira [21] which presented top-down, bottom-up, and Earley parsing approaches to viewing parsing as deduction. One of our aims was to further understand and study this. Some relevant previous work on this topic includes an old approach using Definite Clause grammars (DCGs) also presented by Periera and Warren [17] and its Prolog implementation by Campbell [2]. There is an old paper by Cohen [4] which explains the implementations of some of the algorithms in Prolog that we will later see but his work focuses more on DCGs. Alternately there is an approach using Parsing Expression Grammars (PEGs) as shown by Mascarenhas, Medeiros, and Ierusalimschy [14].

For this paper, we decided to fix a grammar and then study the two different styles of parsing it: Top-Down and Bottom-Up in Prolog. Work to represent grammars has already been done by many people such as Warren [23]. As we will see, some grammars are more naturally suited to certain types of parsers and representations, and some implementations are much more natural in Prolog. In fact, Bottom-Up approaches in general are inefficient for a number of reasons, and people such as Imre [11] have developed other versions of Prolog more suited to this. We will be restricting ourselves to Context-Free-Grammars (CFGs) because of how general and strong they are. Regular expressions, for example, are fast and easy to parse as emphasized by Cox [5] but CFGs are much more interesting as shown by Sikkel [22].

2 Simple Calculator

In this section, we will be showing a mini compiler which will in turn explain why parsing is so important. A compiler for a general parser as would have 3 parts:

- a tokenizer - which converts the atoms to a list of tokens.
- a parser - which parses those tokens and checks whether they follow the grammar.
- a processor and evaluator - after obtaining the parse tree, modern compilers make many rich adjustments where the expression changes but remains the same semantically. This is done to optimize speed and/or time.

Consider the expression:

$$x \times 7 + 61 \times 7$$

It would be tokenized as follows:

```
[num x, sym *, num 7, sym +, num 61, sym *, num 7]
```

The parser would then check whether it follows the grammar, which it does. Then the parse tree would be generated which would look like this:

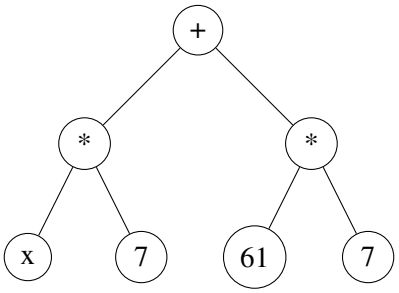

Figure 1: Initial parse tree

An example of a possible simplification modern compilers might make would look like this:

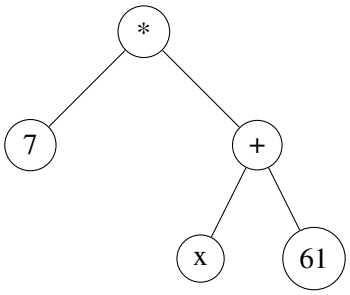

Figure 2: Simplified parse tree

Notice that by doing so we are able to reduce the number of operations by one.

2.1 Tokenizing

The grammar for our calculator is given as follows:

$$
\begin{aligned}
Expr &\rightarrow Lb\ \ Expr\ \ Rb \mid Num \mid Expr\ \ Sym\ \ Expr \\
Num &\rightarrow Minus\ \ Dig\ \ Dig^* \mid Dig\ \ Dig^* \\
Sym &\rightarrow + \mid - \mid / \mid * \\
Minus &\rightarrow - \\
Dig &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
Lb &\rightarrow (\\
Rb &\rightarrow)
\end{aligned}
$$

The following is a simple tokenizer for the above grammar:

```
sym(X):- member(X, ['+', '-', '/', '*']).

% Base case
tokenizer([], Acc, Acc).

% non-empty accumulator and character parsed is a symbol
tokenizer([C|Chars], Acc, Tokens):-
    sym(C),
    append(Acc, [C], NewAcc),
    tokenizer(Chars, NewAcc, Tokens).
```

```
% non-empty accumulator and character parsed is a digit
tokenizer([C|Chars], Acc, Tokens):-
    \+ sym(C),
    atom_number(C, X),
    add_number(Acc, X, NewAcc),
    tokenizer(Chars, NewAcc, Tokens).

% case for empty accumulator
tokenizer([C|Chars], [], Tokens):-
    \+ sym(C),
    atom_number(C, X),
    tokenizer(Chars, [X], Tokens).

% if there already is a number in the accumulator
add_number([Y], X, [Z]):-
    \+ sym(Y),
    Z is Y*10+X.

% There is a symbol in the accumulator
add_number([Y], X, [Y, X]):-
    sym(Y).

% reaching the end
add_number([H|T], X, [H|U]):-
    add_number(T, X, U).
```

By calling the tokenizer, what we obtain is a list of tokens that do not make sense yet and do not need to follow the rules. After tokenizing, we would parse the tokens to make sure they actually follow our grammar. This is what will be further explained in our paper. After making sure it is parsable, we could generate a parse tree which would help us evaluate our expression. Modern compilers would perform many rich simplifications here a simple example of which is shown above.

2.2 Evaluation

With this done, we are able to form our parse trees. In the above figures, we could possibly represent the parse tree as $(x * 7) + (61 * 7)$ for Figure 1 and $7 * (x + 61)$ for figure 2. After this, we can evaluate the innermost bracket at each step and continue until we have a number. The code for that is shown below.

```
%base case
evaluator(PTokens, PTokens):-
    number(PTokens).
%base case
evaluator(['-', PToken2], N):-
    number(PToken2),
    N is ((-1) * Ptoken2).
```

```
% Further simplification is possible
evaluator(PTokens, N):-
    \+ number(PTokens),
    writeln(PTokens),
    evaluation(PTokens, NewPTokens),
    evaluator(NewPTokens, N).

%evaluates based on BODMAS
evaluation([],[]).

evaluation(PTokens, Z):-
    length(PTokens, 1),
    elem_list(PTokens, 0, Z).

evaluation(PTokens, -Z):-
    length(PTokens, 2),
    elem_list(PTokens, 1, Z).

evaluation(PTokens, NewPTokens):-
    member('(', PTokens),
    bhandler(PTokens, NewPTokens).

evaluation([X,'-',Y|PTokens], [Z|PTokens]):-
    \+member('(', PTokens),
    \+member('/', PTokens),
    \+member('*', PTokens),
    \+member('+', PTokens),
    Z is X-Y.

evaluation([X,'+',Y|PTokens], [Z|PTokens]):-
    \+member('(', PTokens),
    \+member('/', PTokens),
    \+member('*', PTokens),
    Z is X+Y.

evaluation([X,'*',Y|PTokens], [Z|PTokens]):-
    \+member('(', PTokens),
    \+member('/', PTokens),
    Z is X*Y.

evaluation([X,'/',Y|PTokens], [Z|PTokens]):-
    \+member('(', PTokens),
    Z is X/Y.

evaluation([P|PTokens], [P|NewPTokens]):-
    evaluation(PTokens, NewPTokens).

% bhandler and stopper: help to handle parenthesis
```

```
%replacein: helps replace the bracketed expression with its value

% elim_list: element at a certain position in the list
```

All the auxiliary functions are mentioned in the appendix.
For the rest of the paper, we will be focusing only on the middle part which is parsing.

3 Top Down vs Bottom Up algorithms

This section will explain the top-down and bottom-up approaches. Let us consider specifying a tree in Prolog with a directed edge between x and y represented by:

```
adj(x,y)
```

If we want to solve the reachability problem then we have a very simple top-down approach, a Depth First Search (DFS):

```
reachTD(X,X)
reachTD(X,Y):-
    path(X,K),
    reachTD(K,Y).
```

Note that the order here is very important. Since Prolog is left-recursive putting $reachTD$ first and $path$ second will make it go in an infinite loop.
The following Bottom-Up implementation takes inspiration from the ideas of Datalog [25]
Firstly, we define our rules to be in the following form: $Rule(A, B)$ where A is inferred from B

```
rule(node(a), []). % repeat for all nodes

rule(adj(a,b), []). %repeat for all edges


rule(path(X,X),[node(X)]).
rule(path(X,Y),[adj(X,Z), path(Z,Y)]).
```

Based on this representation of the rules, we can write the following code:

```
join([], L, L).
join([X|K], L, M):-
    member(X, L),
    !,
    join(K,L,M).
join([X|K],L,[X|M]) :-
    join(K,L,M).

subset([],_).
subset([X|L],K) :-
    member(X,K),
```

```
    subset(L,K).

reachBU(X,Y):-
    length(X, N),
    fchain(X,Y),
    length(Y, N).

reachedBU(X,Y):-
    fchain(X,C),
    length(X, N1),
    length(C, N),
    N \== N1,
    reachedBU(C,Y).

fchain(In, Out):-
    bagof(Atom,Body^(rule(Atom,Body),subset(Body,In)),Conseq),
    join(Conseq, In, Out).
```

One thing to be noted is that $reachTD$ and $reachBU$ take different inputs. $reachTD$ takes in just the two nodes to check whether they are connected while $reachedBU$ takes an empty list and a variable. It keeps on adding elements to the list to saturate it and in the end, we can check if our required path is in the list.

The Bottom-Up code maintains a list of all nodes that have been reached (it is initialized with the empty list), and at each step, it saturates the list by adding members that did not exist before in the list. But, these new members can be inferred from the current ones. The idea is to keep checking if elements can be added and add them until the length of the list does not change. The following are some important observations.

- The top-down approach might not work for graphs with cycles however the bottom-up approach will work regardless. This is because the top-down approach might get stuck in a recursive loop but in the bottom-up approach we check if something does not already exist before adding it avoiding the infinite loop. Hence while top-down approaches are limited to Acyclic graphs, Bottom-up approaches are not.
- Once the list has been generated for the bottom-up approach, there is no need to re-infer a path. Instead, we can check the same list. Hence this technique is used very commonly in finite databases where the parameters do not change a lot. The most prominent example of this is Datalog [25].
- On the opposing side to the second point, if a single query needs to be made then the top-down approach will be faster since it does compute reachability only for the goal. This is easily seen in cases of early termination when the nodes are near the root. However, both of them are $\mathcal{O}(E)$ where E denotes the number of edges.
- Lastly, top-down is much easier to write.

Both approaches have a space complexity of $\mathcal{O}(V^2)$ where V is the number of vertices. The list in the bottom-up approach keeps on growing linearly. On the other hand, for the top-down approach one must maintain the stack for the path taken to potentially backtrack and choose a different path in case one does not work out.

4 Parsing Algorithms and Grammars

It is a well-known fact that every Context-Free-Grammar can be expressed in Chomsky Normal Form (CNF) [3] wherein every rule has one of three forms:

$$A \rightarrow BC \quad or$$
$$A \rightarrow a \quad or$$
$$S \rightarrow \epsilon$$

The last condition is that the start symbol cannot appear on the right-hand side of any rule. (It is to be noted that only the start symbol can have an epsilon production rule) We also know about Greibach Normal Form (GNF) [9] where every rule is of the form :

$$A_0 \rightarrow aA_1A_2A_3...A_n$$

for some $n \in \mathbb{N}$ (including zero) which is also equivalent to a CFG. If we remove all the epsilon production rules then we increase the size of the grammar but make the code analysis easier. This is known as the strict Greibach Normal Form. By ensuring this, we can still consider all the CFGs that do not produce the empty word. So instead of the typical Dyck Language given by $G_1 =$

$$S \rightarrow (S)S \mid \epsilon$$

We can consider $G_2 =$

$$S \rightarrow (S)S \mid (S) \mid ()S \mid ()$$

In fact, $L[G_2] = L[G_1] \setminus \{\epsilon\}$ where $L[G]$ represents the language generated by the grammar G. The Dyck Language in fact is one of the most common examples we will use to test our programs and also show the correctness of a program. The following is the CNF representation used:

$$T \rightarrow L_2R_2 \mid L_1R_2 \mid L_2R_1 \mid L_1R_1 \mid \epsilon$$
$$S \rightarrow L_2R_2 \mid L_1R_2 \mid L_2R_1 \mid L_1R_1$$
$$L_2 \rightarrow L_1S$$
$$R_2 \rightarrow R_1S$$
$$L_1 \rightarrow ($$
$$R_1 \rightarrow)$$

and the following is the GNF representation

$$S \rightarrow (RS \mid (R \mid (SR \mid (SRS$$
$$R \rightarrow)$$

Some other interesting grammars that we tested on can be found in the appendix.

4.1 Stack Based Parsing

The first algorithm we will look at is the Stack Based Parser which is a Top Down Approach. It is very well suited to Prolog, especially in Greibach Normal Form. Firstly the grammar is represented as a 3-tuple (X, Y, Z) where X (1 has been reserved for the start symbol) represents the identifier for a rule, Y represents the terminal symbol, and Z is a possibly empty list of the non-terminal symbols. Then, the code is relatively small, and it looks like this:

```
%base case
parse_rule([], []).

%parses according to the rules in the stack
parse_rule([C|Str], [N|Ns]):-
    rule(N,C,Ans),
    append(Ans, Ns, NewL),
    parse_rule(Str, NewL).
```

```
% main function
main(X):-
    atom_chars(X,Y),
    % 1 is the start symbol
    parse_rule(Y, [1]),
    writeln('Parsable').
```

The above-presented algorithm while super simple and natural, has a complexity of $\mathcal{O}(|g|n^n)$ where $|g|$ represents the size of the grammar and n represents the size of the string to be parsed. The key observation here is that, since we are in Greiback Normal Form, we are sure that at each iteration the size of the string to be parsed will decrease by one, and that we will not be considering more non-terminals than the length of the word itself.

4.1.1 Look Ahead and Dynamic Programming

In the above algorithm, not only is the backtracking very inefficient, but it also does not help us produce useful error messages. How will we know where and what the problem is? Both of these can be helped by using lookahead features. We are able to make this algorithm faster with the same rule representation, but it does increase how complicated and long the program is. The idea here is that we look at the next letter and then go through only those rules that match that letter. This is seen in the splitter function, in the implementation below, which accordingly splits the string as two substrings to be parsed.

```
%base case
parse([], []).

%parses according to the rules in the stack
parse(Str, [L, M|Ls]):-
    rule(M, A, _,),
    member(A, Str),
    splitter(Str, S1, S2, A),
    parse(S1, [L]),
    parse(S2, [M|Ls]).

parse([C|Str], [L]):-
    rule(L, C, Li),
    parse(Str, Li).

% main function is the same

splitter([Split|Str], [], [Split|Str], Split).

splitter([S|Str], [S|Subone], Subtwo, Split):-
    splitter(Str, Subone, Subtwo, Split).
```

While this makes the code faster in practice, in theory, the complexity remains the same. With this, we can see that this top-down approach is very easy to implement but impractical. The main thing to learn from this is that the large complexity is due to the non-determinism of the parsing. Reducing this will reduce the

complexity. Lastly, one often overlooked advantage of this approach is that we parse the string from left to right (this is not always true - for example, we will see the CYK algorithm later). Since this is how we have designed our languages, and it works in one pass it is also good for finding ambiguities in our grammar. In fact, efficient parsing goes hand in hand with detecting ambiguities in a grammar as we will see later in the Earley Parsing section. [18] [19] [7].
Lastly, this idea is very widely used in modern-day compilations (for programming languages) except instead of CFGs, we use a restricted grammar known as $LALR(1)$ which means that not only is our grammar deterministic, but also every rule will have a different first letter. This makes parsing much faster and deterministic as shown by DeRemer and Pennello [6].

4.2 CKY Parsing

Now that we have seen the Top-Down approach let us look at the bottom-up approach called the Cocke–Younger–Kasami (CKY or CYK) algorithm [29] mentioned in most of our references.
Our traditional implementation of the algorithm, in Python, can be seen below.
The grammar here has been taken in Chomsky Normal Form for reasons we will see later. It looks like this:

```
    grammar = {
    'S': {'AB', 'AC', 'AD', 'ED'},
    'A': {'('},
    'B': {')'},
    'C': {'AC'},
    'D': {'CA'},
    'E': {'BA'}
}
```

and the code is as follows:

```
    def printer(arr):
    width = max(len(str(element)) for row in arr for element in row)
    for row in arr:
        for element in row:
            print(str(element).rjust(width), end=" ")
        print()
    print()

def cyk(grammar, start_symbol, string):
    n = len(string)
    table = [[set() for _ in range(n)] for _ in range(n)]
    # Fill in the base cases of the CYK table
    for i in range(n):
        for rule in grammar:
            if string[i] in grammar[rule]:
                table[i][i].add(rule)
    # Fill in the rest of the CYK table
    for length in range(1, n):
        printer(table)
        for i in range(n - length):
            j = i + length
            for k in range(i, j):
```

```
                for rule in grammar:
                    for rule1 in table[i][k]:
                        for rule2 in table[k+1][j]:
                            if rule1 + rule2 in grammar[rule]:
                                table[i][j].add(rule)
    # Check if the start symbol is in the top-right corner of the table
    return start_symbol in table[0][n - 1]
```

The idea of this algorithm is to create a table of size $n \times n$ where n represents the length of the string. Then at position $(i, j) \in \mathbb{N} \times \mathbb{N}$ it fills up all the symbols which can parse the substring from position i to j. Since it is a bottom-up algorithm, it starts with just the parsing of every individual character in the string, and when this is over it parses every substring of length 2. Then every substring of length 3 until it eventually parses the entire string. Since we have restricted ourselves to the Chomsky Normal Form we know that if we want to parse a string between i and j then either there is a single non-terminal between them or there exists a $k \in \mathbb{N}, k > i$ and $k < j$ such that, we, in the table, have previously seen a rule (say A) parse (i, k) and another (say B) parse (k, j) then we can look for all rules whose right-hand sides are AB. It is because of this and the reduced non-determinism that the algorithm would work in $\mathcal{O}(|g|n^3)$ where $|g|$ represents the size of our grammar and n represents the size of the string to be parsed.
Notice that we do not need any other property of the CNF other than the fact that it has a maximum of two non-terminal symbols together. In general, for a grammar with a maximum of $k \in \mathbb{N}$ non-terminals on its right-hand side, we can use the CYK to parse it in $\mathcal{O}(|g|n^{k+1})$. The algorithm has been widely criticized in papers by people such as Ullman [1] for still being slow and too heavy on space; however, small modifications made to the algorithms such as those presented by Lange [12] seem to take care of most issues.

This is the standard implementation of this algorithm. However, there is no natural way to build this table in Prolog. Hence we have come up with a more Prolog-esque implementation of the same algorithm with the same complexity. ALl auxiliary functions can be found in the appendix. The main code can be seen below:

```
%pos(L, I, J, S):- a function to find the element S between i and j
                   in L can be seen in the appendix

main(X,C,L):-
    closure(X,C),
    %X and C have the same length then our algorithm is over
    member((0,L,sent),C).

main(X,Y,L):-
    closure(X,C),
    %C has a larger length we need to further saturate
    main(C,Y,L).

closure(X, [(I,J,np)|X]):-
    nP(I,J,X),
    \+ member((I,J,np), X).

closure(X, [(I,J,vp)|X]):-
    vP(I,J,X),
    \+ member((I,J,vp), X).

% ... repeat for all other grammar rules
```

```
closure(X,X).
```

here the grammar is represented by prolog clauses of the form :

```
% As seen in Shieber
senT(I,J,S):-
    member((I,K,np), S),
    member((K, J, vp), S).
```

or

```
nP(I,J,S):-
    pos(S,I,J,'Trip'),
    J is I+1.
```

or a combination of these.

A full grammar for this can be seen in the appendix. We can see that the implementation is in fact not far from the bottom-up implementation presented earlier. We constantly add symbols based on the deductive closure before increasing the size and proceeding with that.
Lastly, we would like to emphasize that the grammars in this format are equivalent to the CFGs. Since we know that grammars in CNF are equivalent to CFGs, we shall show the equivalence to our grammars in CNF by writing a small program to convert one to the other. First is the conversion from CNF into the above form:

```
toCNF():-
    rule(N, A, B),
    write(N),
    writeln('(I,J,S):-'),
    write('member((I, K, '),
    write(A),
    writeln('), S),'),
    write('member((K, J, '),
    write(B),
    write('), S).').

toCNF():-
    rule(N, A),
    write(N),
    writeln('(I,J,S):-'),
    write('pos(S,I,J,'),
    write(A),
    writeln('),'),
    writeln('J is I+1.').
```

Similarly, on the other side,

```
    rule(predicate_name, A)
```

where A is the last argument given to the pos function.

```
rule(predicate_name, A, B)
```

where A is the last argument of the first member and B is the last argument of the second.

We have shown a short and natural implementation of the CKY algorithm in Prolog. The idea here is that we start with a list of just the sentence as atoms and at each step, we see what we can infer from there. It is the same idea of saturation as we have seen before. Once saturated the algorithm automatically tries bigger and bigger intervals and fills them up until the whole string is parsed or not. In the end, we simply need to check if $sent(1, L)$, where L is the length of our string, belongs to the list or not. What this shows is that if you change the nature of inference then the nature of CYK is more free - By computing the deductive closure of the set of facts under the rules we achieve the same end result of looking at increasing size intervals.

5 Correctness

A good parser has the following qualities. [15]

- soundness - every parse is valid
- guaranteed termination
- complete - for any sentence the parser is sound

We wanted to make sure our parser had these qualities.

5.1 Soundness and Completeness

A way to verify that all our previous algorithms are correctly implemented is to generate a Parse Tree for the words we have parsed. A parse tree is used to see which rules are applied in what order and which rules they branch out to. More formally, a parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. The term parse tree itself is used primarily in computational linguistics; in theoretical syntax, the term syntax tree is more common [24].
To that end, we built a Prolog program that given a grammar generates all possible parse trees. The challenge in this segment was that Prolog is implemented such that its search descends in the left-most branch until it reaches a node. Hence we came up with the following implementation shown on the previous bracketing example.

```
%base case
parse_rule([], '').

%parses according to the selected rule
parse_rule(['('|Str], a(X,Y)):-
    parse_subword([2], Str, NParse1, Parse1),
    parse_subword([1], NParse1, [], NParse1),
    parse_rule(Parse1, X),
    parse_rule(NParse1, Y).

parse_rule(['('|Str], b(X)):-
    parse_subword([2], Str, [], Str),
    parse_rule(Str, X).

parse_rule(['('|Str], c(X,Y)):-
    parse_subword([1], Str, NParse1, Parse1),
```

```
    parse_subword([2], NParse1, [], NParse1),
    parse_rule(Parse1, X),
    parse_rule(NParse1, Y).

parse_rule(['('|Str], d(X,Y,Z)):-
    parse_subword([1], Str, NParse1, Parse1),
    parse_subword([2], NParse1, NParse2, Parse2),
    parse_subword([1], NParse2, [], NParse2),
    parse_rule(Parse1, X),
    parse_rule(Parse2, Y),
    parse_rule(NParse2, Z).

parse_rule([')'], e).

% main function
main(X, ParseTree):-
    atom_chars(X,Y),
    parse_rule(Y, ParseTree).

parse_subword([], Input, Input, []).
parse_subword([R|Rules], [I|Input], Output, [I|Parsed]):-
    rule(R, I, NTs),
    append(NTs, Rules, NewRules),
    parse_subword(NewRules, Input, Output, Parsed).
```

At each step, we find how to break the string by parsing all possible combinations of substrings in order to counter the recursive proof search of Prolog. Indeed, with the input:

```
    main("(())", X).
```

one would obtain the expected results of $X = c(b(e), e)$ where the letters represent the rules in lexicographic order.

5.1.1 Automatic Parse Tree

Writing such programs is highly systematic and needs to be precise. Hence, Programs known as parse generators are commonly used to automate and be precise in this task. In this, CFGs need to be input in a particular format and the result is a program that produces proof trees like the one above. This will also result in a better understanding of our original goal as can be seen later in this section. Our program deals with Grammars in GNF and each rule is also given a unique character to identify it.

```
%main function to write the main and parse-tree functions and then call the functio
main

%Deal with rules with no non-terminals
rulemaker(B,[],D):-
    write('parse_rule([\''),
    write(B),
    write('\'],'),
    write(D),
```

```
    writeln(').').

%Deal with rules with 1 non-terminal
rulemaker(B, C, D):-
    length(C, A),
    A>0,
    write('parse_rule([\''),
    write(B),
    write('\'|Str],'),
    write(D),
    write('('),
    write_letters(C, 65, Output),
    writeln(')):-'),
    write_subwords(C, 1),
    write_parse(1, Output).

% To write the letters and make a list of them
write_letters([_], Num, [Char]):-
    char_code(Char, Num),
    write(Char).
write_letters([_|Ch], Num, [Char|Output]):-
    length(Ch, A),
    A>0,
    char_code(Char, Num),
    write(Char),
    write(','),
    Num2 is Num+1,
    write_letters(Ch, Num2, Output).

% To write the correct number of parse_subword functions depending on non-%termina
write_subwords([C], N):-
    write('    parse_subword(['),
    write(C),
    write('], '),
    write('NParse'),
    N2 is N-1,
    write(N2),
    write(', [], '),
    write('NParse'),
    write(N2),
    writeln('),').

write_subwords([C|Ch], 1):-
    length(Ch, A),
    A>0,
    write('    parse_subword(['),
    write(C),
    writeln('], Str, NParse1, Parse1),'),
    write_subwords(Ch, 2).
```

```
write_subwords([C|Ch],N):-
    N \==1,
    length(Ch, A),
    A>0,
    write('    parse_subword(['),
    write(C),
    write('], '),
    write('NParse'),
    N2 is N-1,
    write(N2),
    write(', NParse'),
    write(N),
    write(', '),
    write('Parse'),
    write(N),
    writeln('),'),
    N3 is N+1,
    write_subwords(Ch, N3).

% To correctly write the parse functions that recursively generate trees
write_parse(N, [O]):-
    write('    parse_rule(NParse'),
    N2 is N-1,
    write(N2),
    write(', '),
    write(O),
    writeln(').').

write_parse(N, [O|Output]):-
    length(Output, A),
    A>0,
    write('    parse_rule(Parse'),
    write(N),
    write(', '),
    write(O),
    writeln('),'),
    N2 is N+1,
    write_parse(N2, Output).
```

One observes that upon entering the previous grammar here, the previous program is spit out.

5.2 Guaranteed Termination

We notice that for each of our algorithms, termination is guaranteed. This is because in the top-down approach, we always decrease the size of the string by one, and in the bottom-up approach we choose a finitely small substring to parse based on what was already parsed.

6 Future Work

6.1 Earley Algorithm

The best algorithm we have seen so far is the CKY algorithm and as seen it runs in $\mathcal{O}(n^{k+1})$ where k represents the number of non-terminals in the rule with the maximal number of non-terminals. Hence, by constraining ourselves to CNF we can obtain a cubic algorithm. However, this obviously does not work for all CFGs. Instead, Earley had given an algorithm [7] which runs in $\mathcal{O}(n^3)$ for any CFG. Furthermore, for unambiguous grammars, it runs in $\Omega(n^2)$ and for deterministic grammars, it runs in $\Omega(n)$ [26]
We implemented the algorithm in Python below.

```
class State:

    def __init__(self, path, rule, i, origin):
        self.rule = rule
        self.path = path
        self.i = i
        self.origin = origin
        self.parse = path[:i]
        self.nparse = path[i:]

    def __eq__(self, other):
        return  (self.rule == other.rule and self.path == other.path and
                 self.i == other.i and self.origin == other.origin)

    def __hash__(self):
        return hash((self.rule, self.path, self.i, self.origin))

    def canComplete(self):
        return self.i == len(self.path)

    def canPredict(self):
        return (not self.canComplete()) and self.nparse[0].isdigit()

    def canScan(self, symbol):
        return self.nparse and (not self.canComplete()) and
               self.nparse[0] == symbol

def parse(st):
    chart = [set() for _ in range(len(st)+1)]
    for path in grammar['0']:
        chart[0].add(State(path, 0, 0, 0))
    k=0
    while k < (len(st)):
        l1 = len(chart[k])
        predict(k, chart)
        complete(k, chart)
        if(len(chart[k])>l1):
            continue
        scan(st[k], k, chart)
```

```
        k+=1
    for state in chart[len(st)]:
        if state.canComplete and state.rule == 0:
            return True
    return False

def predict(i, chart):
    k=0
    while(k<len(chart[i])):
        for k in range(k,len(chart[i])+1):
            if(k==len(chart[i])):
                break
            state = list(chart[i])[k]
            if state.canPredict():
                next = state.nparse[0]
                for path in grammar[next]:
                    chart[i].add(State(path, int(next), 0, i))

def scan(symbol, i , chart):
    for state in chart[i]:
        if state.canScan(symbol):
            chart[i+1].add(State(state.path, state.rule, state.i+1, state.origin))

def complete(i, chart):
    k=0
    while(k<len(chart[i])):
        for k in range(k, len(chart[i])+1):
            if(k==len(chart[i])):
                break
            state = list(chart[i])[k]
            if state.canComplete():
                for estate in chart[state.origin]:
                    if estate.nparse and estate.nparse[0] == str(state.rule):
                        chart[i].add(State(estate.path, estate.rule,
                                    estate.i+1, estate.origin))
```

The interesting part about this algorithm itself is that it combines aspects of both Top-Down and Bottom-Up aspects and there is no obvious simple and natural implementation of it in Prolog. The interesting part about the implementation is that the size of the set changes during the iteration. Since the compilers for most languages don't allow this we must come up with a different approach. Hence we ran a loop to check if the size of the set increased and then looped only over the new elements. Lastly, it is important to run the three functions: scan, complete, and predict over the newly added elements as well.

The grammar dictionary might look something like:

```
grammar = {
'0':['(0)0', '()', '(0)', '()0']
}
```

One key aspect to be noted is that the grammar still remains the same and is in fact very close to our Prolog

representations. So we started out by trying to fix CFGs and trying to parse them in the two different approaches. However, this has shown that the same CFG can also be parsed via a third approach which is a combination of those two. A future aspect remains to better understand the algorithm, the way it combines the two approaches, and implementing it naturally in Prolog.

6.2 Better Parsing Algorithms

As seen above, for any Context-Free-Language Earley [7] had given an algorithm which combined bottom-up and top-down approaches to yield an algorithm which runs in $\mathcal{O}(|g|n^3)$ where $|g|$ represents the size of the grammar and n represents the size of the string being parsed. It was later shown by Ebert, [8] building on the works by Lee, [13] that parsing a Context-Free-Grammar can be reduced to the multiplication of Boolean Matrices which we know is sub-cubic. In fact, as Williams [28] showed earlier this year, the best algorithm we have runs in $\mathcal{O}(n^{2.371552})$. However, the lowest theoretical possible complexity remains an open question. This leaves us with two interesting problems based on what we have seen.

1. Is there a better combination of the bottom-up and top-down approaches in the style of the matrix multiplication algorithms which are faster than Earley's algorithm> In fact, researchers such as Satta [20] are currently have published papers on this topic.

2. What is the theoretically lowest possible complexity for parsing any CFG?

In practicality, CFG parsers are not very widely used, and as mentioned earlier LALR(1) parsers are used to make compilations subquadratic. Further, when we do need to parse a CFG, the inputs generally are not big enough for us to substantially notice a difference between the Earley and better algorithms.

6.3 Term Sharing and Common SubFormula Elimination

Consider the graphs below

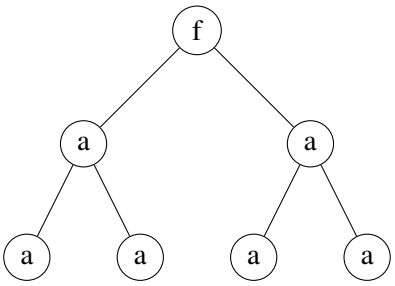

Figure 3: Without Sharing

and let us compare it to

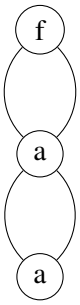

Figure 4: With Sharing

Observe that we can the latter is just another representation of the former however Graph 1 grows exponentially while Graph 2 will grow only linearly. The second graph uses Sharing.

Now let us look at two different parse trees

```
f(a(b(c,d), e), g(b(c,d)))
```

and

```
f(a(n,e), g(n)) where n = b(c,d)
```

While this example is not as drastic as the first one, one can easily see it may make a huge difference. This is known as common sub-formula elimination and is a technique commonly used by modern compilers.

One way to help do it is to bring up to each node the symbols of the roots below. This can be shown by the following example:

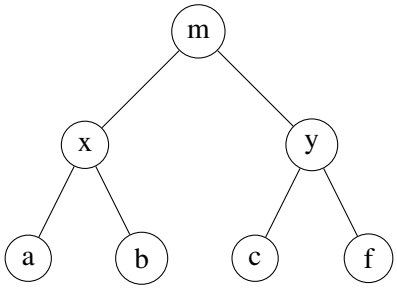

Figure 5: easy example of a tree

We could then do the following representation

The code for this is significantly trickier as our parse tree representation is not very parsable and we are not restricted to binary trees. Hence at each step, we must convert to a list and apply this operation for every possible element in the list. To simplify it, we show it only for a particular depth

```
depth(S, 0, Ans):-
    rbracket(S,Ans).
depth(S, D, Ans):-
    layer1(S,S1),
    D2 is D-1,
```

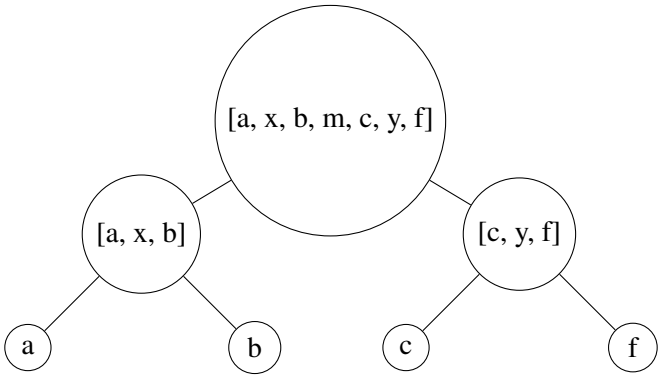

Figure 6: tree with the leaves brought up

```
    \+ member(',', S1),
    depth(S1,D2,Ans).
depth(S, D, Ans):-
    layer1(S,S1),
    D2 is D-1,
    member(',', S1),
    listmaker(S1, L),
    writeln(L),
    depth_list(L, D2, Ans).

depth_list([], _, []).
depth_list([L|Ls], D, [A|Ans]):-
    depth(L, D, A),
    depth_list(Ls, D, Ans).

listmaker([], []).
listmaker(S, [S1|L]):-
    firststring(S, S1),
    writeln(S1),
    append(S1, [','|S2], S),
    listmaker(S2, L).
listmaker(S, [S]):-
    firststring(S, S).

%first string returns the substring up to the first,

%butlast, rbracket, layer1, and layer2 help us eliminate the first letter and brac


main(S, D, Ans):-
    atom_chars(S,X),
    depth(X, D, Ans).
```

The bottom-up approach we have seen is beneficial in implementing solutions like this. Notice, that in our

prolog implementation of the CKY algorithm, we saturate the list with all possible subwords of a certain length before increasing this length. Then we can easily check if any of these subwords are disjunct and the same in which case they can be shared or represented differently.

7 Conclusions

There has been a lot learned in this project about CFGs, Normal Forms, complexity issues in Parsing, Top-Down and Bottom-Up approaches, parsing in Prolog with them, different parsing algorithms, and their connections with logic and deduction. Based on the existing literature we have tried to show that we can fix a grammar and parse it top-down, bottom-up, or as a mixture of both. Furthermore, as we have seen, we have managed to make natural implementations of the Bottom-Up approach in Prolog for the CYK algorithm and this has shown us that by changing the nature of inference we have made the CKY algorithm more free- By computing the deductive closure of the set of facts under the rules we achieve the same end result of looking at increasing size intervals and computing the parsability of a string.

8 References

[1] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., USA, 1972.

[2] J. A. Campbell. *Implementations of prolog*. John Wiley And Sons, 1984.

[3] Noam Chomsky. On the representation of form and function. *The Linguistic Review*, 1981.

[4] Jacques Cohen and T. Hickey. Parsing and compiling using prolog. *ACM Trans. Program. Lang. Syst.*, 9:125–163, 1987.

[5] Russ Cox. Regular expression matching can be simple and fast, 2007.

[6] Frank DeRemer and Thomas Pennello. Efficient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):615–649, 1982.

[7] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, feb 1970.

[8] Franz Christian Ebert. Cfg parsing and boolean matrix multiplication. Accesed March, 2007.

[9] Sheila A Greibach. A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM (JACM)*, 12(1):42–52, 1965.

[10] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[11] Imre Kilián. Contralog: a prolog conform forward-chaining environment and its application for dynamic programming and natural language parsing. *Acta Universitatis Sapientiae, Informatica*, 8, 02 2016.

[12] Martin Lange and Hans Leiss. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didact.*, 8, 2009.

[13] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication, 2001.

[14] Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. On the relation between context-free grammars and parsing expression grammars. *Science of Computer Programming*, 89:235–250, 2014.

[15] Andrew McCallum. Introduction to natural language processing, 2004.

[16] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, USA, June 1983. Association for Computational Linguistics.

[17] Fernando C.N. Pereira and David H.D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[18] Frank Pfenning. Lecture notes on context-free grammars. [Online; accessed 10-March-2024].

[19] Frank Pfenning. Lecture notes on top-down predictive ll parsing. [Online; accessed 10-March-2024].

[20] Giorgio Satta and Oliviero Stock. Bidirectional context-free grammar parsing for natural language processing. *Artificial Intelligence*, 69(1):123–164, 1994.

[21] Stuart M. Shieber, Yves Schabes, and Fernando C.N. Pereira. Principles and implementation of deductive parsing. *The Journal of Logic Programming*, 24(1):3–36, 1995. Computational Linguistics and Logic Programming.

[22] Klaas Sikkel and Anton Nijholt. *Parsing of Context-Free Languages*, pages 61–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[23] David S. Warren. *Programming in Tabled Prolog*. SUNY @ Stony Brook, 1999.

[24] Wikipedia contributors. Parse tree — Wikipedia, the free encyclopedia, 2023. [Online; accessed 10-March-2024].

[25] Wikipedia contributors. Datalog — Wikipedia, the free encyclopedia, 2024. [Online; accessed 16-March-2024].

[26] Wikipedia contributors. Earley parser — Wikipedia, the free encyclopedia, 2024. [Online; accessed 15-March-2024].

[27] Wikipedia contributors. Parsing — Wikipedia, the free encyclopedia, 2024. [Online; accessed 16-March-2024].

[28] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega, 2023.

[29] Daniel H Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967.

A Appendix

A.1 Other grammars used

A.1.1 Even number of 1s in a binary string

- Regular expression

$$0^*(10^*10^*)^*$$

- CNF

$$\begin{aligned}
S &\rightarrow ZA \\
Z &\rightarrow OZ \mid \epsilon \\
O &\rightarrow 0 \\
A &\rightarrow LA \mid \epsilon \\
L &\rightarrow TT \\
T &\rightarrow BZ \\
B &\rightarrow 1
\end{aligned}$$

- GNF

$$\begin{aligned}
Z &\rightarrow oZ \mid o \\
S &\rightarrow oS \mid 1C \mid 1ZC \\
B &\rightarrow 1 \\
C &\rightarrow 1S \mid 1ZS \mid 1 \mid 1Z \mid 0BZ \mid 0ZB \mid 0ZBZ \mid 0B \mid 0BZS \mid 0ZBS \mid 0ZBZS \mid 0BS
\end{aligned}$$

A.1.2 Binary Strings ending in 0011

- Regular expression

$$(0+1)^*0011$$

- CNF

$$\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow ZA \mid OA \mid \epsilon \\
B &\rightarrow Z'O' \\
Z' &\rightarrow ZZ \\
O' &\rightarrow OO \\
Z &\rightarrow 0 \\
O &\rightarrow 1
\end{aligned}$$

- GNF

$$\begin{aligned}
S &\rightarrow 0AB \mid 1AB \mid 0ZOO \\
A &\rightarrow 0A \mid 1A \mid 0 \mid 1 \\
B &\rightarrow 0ZOO \\
Z &\rightarrow 0 \\
O &\rightarrow 1
\end{aligned}$$

A.2 CYK Algorithm

A.2.1 Auxillary functions for

```
pos([(I, J, S)|_], I, J, S).

pos([(K, _, _)|Atoms], I, J, S):-
    K \== I,
    pos(Atoms, I, J, S).

pos([(_, K, _)|Atoms], I, J, S):-
    K \== J,
    pos(Atoms, I, J, S).
```

A.2.2 Full grammar for the Prolog example

```
    %taken from Scheiber
senT(I,J,S):-
    member((I,K,np), S),
    member((K, J, vp), S).

nP(I,J,S):-
    det(I,K,S),
    K is I+1,
    no(K,L,S),
    L is K+1,
    member((L, J, optrel), S).

nP(I,J,S):-
    det(I,K,S),
    K is I+1,
    no(K,J,S),
    J is K+1.

nP(I,J,S):-
    pos(S,I,J,'Trip'),
    J is I+1.

nP(I,J,S):-
    pos(S,I,J,np).

vP(I,J,S):-
    tV(I,K,S),
    K is I+1,
    member((K, J, np), S).

vP(I,J,S):-
    pos(S,I,J,'swings'),
    J is I+1.
```

```
vP(I,J,S):-
    pos(S,I,J,vp).

optRel(I,J,S):-
    relPro(I,K,S),
    K is I+1,
    member((K, J, vp), S).

optRel(I,J,S):-
    pos(S,I,J,optrel).

det(I,J,S):-
    pos(S,I,J,'a'),
    J is I+1.

no(I,J,S):-
    pos(S,I,J,'lindy'),
    J is I+1.

tV(I,J,S):-
    pos(S,I,J,'dances'),
    J is I+1.

relPro(I,J,S):-
    pos(S,I,J,'that'),
    J is I+1.
```

A.3 Simple calculator CNF Prolog grammar

A.3.1 Grammar in CNF

```
rule(1, 2, 3).
rule(1, 4, 0).
rule(1, 1, 5).
rule(3, 1, 6).
rule(5, 7, 1).
rule(4, 0, 10).
rule(4, 9, 10).
rule(10, 8, 10).
rule(10, 8, 0).

rule(0, '').
rule(2, '(').
rule(6, ')').
rule(7, '+').
rule(7, '-').
rule(7, '/').
rule(7, '*').
```

```
rule(9, '-').
rule(8, '0').
rule(8, '1').
rule(8, '2').
rule(8, '3').
rule(8, '4').
rule(8, '5').
rule(8, '6').
rule(8, '7').
rule(8, '8').
rule(8, '9').
```

A.3.2 Auxilliary functions

```
%help find the bracketed expression to simplify
stopper([')'|_], [], 1).
stopper(['('|PTokens], ['('|Exp], C):-
    stopper(PTokens, Exp, C+1).
stopper([')'|PTokens], ['('|Exp], C):-
    stopper(PTokens, Exp, C-1).
stopper([P|PTokens], [P|Exp], C):-
    stopper(PTokens, Exp, C).

%replace the bracketed expression
replacein([')'|NewPTokens], NewPTokens, 1).
replacein(['('|PTokens], NewPTokens, C):-
    replacein(PTokens, NewPTokens, C+1).
replacein([')'|PTokens], NewPTokens, C):-
    replacein(PTokens, NewPTokens, C-1).
replacein([_|PTokens], NewPTokens, C):-
    replacein(PTokens, NewPTokens, C).

bhandler(['('|PTokens], [N|NewPTokens]):-
    stopper(PTokens, Exp, 1),
    writeln(Exp),
    evaluator(Exp, N),
    replacein(PTokens,NewPTokens, 1).

bhandler([P|PTokens], [P|NewPTokens]) :-
    P \== '(',
    bhandler(PTokens, NewPTokens).

elem_list([L|_], 0, L).
elem_list([_|List], N, Elem):-
    elem_list(List, N-1, Elem).
```

A.4 Auxilliary functions in Automatic Parse Tree generator

```
    main():-
    writeln('main(X, ParseTree):-'),
    writeln('   atom_chars(X,Y),'),
    writeln('   parse_rule(Y, ParseTree).'),
    writeln(''),
    writeln('parse_subword([], Input, Input, []).'),
    writeln('parse_subword([R|Rules], [I|Input], Output, [I|Parsed]):-'),
    writeln('   rule(R, I, NTs),'),
    writeln('   append(NTs, Rules, NewRules),'),
    writeln('   parse_subword(NewRules, Input, Output, Parsed).'),
    writeln(''),
    rule(_,B,C,D),
    rulemaker(B, C, D).
```

A.5 Auxilliary functions in Future Work

```
firststring([','|_], []).
firststring([C|Ch], [C|Ans]):-
    C \== ',',
    firststring(Ch, Ans).

butlast([')'], []).
butlast([C|Ch], [C|Ans]):-
    length(Ch, N),
    N>0,
    butlast(Ch, Ans).

rbracket([], []).
rbracket([C|Cs],[C|Ans]):-
    C \== '(',
    C \==')',
    C \== ',',
    writeln([C|Cs]),
    rbracket(Cs, Ans).

rbracket(['('|Cs],Ans):-
    rbracket(Cs, Ans).

rbracket([')'|Cs],Ans):-
    rbracket(Cs, Ans).

rbracket([','|Cs],Ans):-
    rbracket(Cs, Ans).

layer1([_|Ss], Ans):-
```

```
    layer2(Ss, Ans).

layer2(['('|Ch], Ans):-
    butlast(Ch, Ans).
```