BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

CYK Algorithm Adapted to the Penttonen Normal Form

Dominika Klobučníková*

Abstract
This paper deals with the topic of context-sensitive grammars as special cases of unrestricted grammars, their normal forms and applications of these normal forms. It presents an algorithm designed using the principles of Cocke-Younger-Kasami algorithm to make a decision, whether an input string is a sentence of a context-sensitive grammar. The final application, which implements this algorithm, works with context-sensitive grammars in Penttonen normal form.

Keywords: formal languages — normal forms — Kuroda — Penttonen — syntax analysis — CYK — CKY — Cocke-Younger-Kasami

Supplementary Material: Theoretical Example (http://bit.ly/2q61uu5)

*xklobu01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Syntax analysis is an essential component of many disciplines focusing on string processing – one such field is program interpretation. In practice, it has always been limited to unambiguous, context-free grammars, as the less restricted families of grammars had proven both hard to use and too difficult to process. However, as the aforementioned area of study has been progressing, the context-freeness offered by this set of grammars became no longer satisfactory, and a need to decompose the input string in a way reflecting natural language has arisen. Context-sensitive grammars, and therefore languages generated by them, present a suitable interlink between these sets, because they allow for context, which is an important aspect of human speech. Currently, very few parsers capable of detecting context, and therefore of processing grammars of this family, exist. The aim of this paper is to design and implement such an algorithm that can deterministically decide, whether the input string is a sentence generated by some grammar. To allow for maximum flexibility, the grammar is to be specified by the user.

The algorithm introduced in this paper is based on Cocke-Younger-Kasami (CYK) parsing algorithm for context-free grammars [1], which works with grammars in Chomsky normal form. To decide whether a string is a member of some grammar, it considers all derivation sequences possibly leading to the desired string. It works in a bottom-up way.

The presented algorithm works with grammars in Penttonen normal form, as an expansion of Chomsky normal form. Apart from the way CYK works, it also applies a set of restrictions to prevent inconsistencies of the derivation tree. If some nonterminal was used to apply both a context-sensitive and a context-free rule, it could lead to occurences of nonterminals that would not appear in the derivation tree under normal conditions. In event of such rule collision, the syntax tree is split into two versions, of which each propagates a different rule. In the event of failure of one such tree, a substitute is chosen and the process is repeated until the string is either accepted, or there are no other possible syntax trees to be examined and the string is rejected.

This algorithm offers the ability to parse an input string according to any user-specified context-sensitive grammar in Penttonen normal form [2]. Since every context-sensitive grammar can be converted to an equivalent grammar in Penttonen normal form, the algorithm therefore works for any such grammar. This algorithm always halts even for cyclic and ambiguous

grammars. Details of its functionality will be further described in the following sections.

2. Unrestricted Grammars

Grammar is an ordered pair, $G = (\Sigma, R)$, where:

- Σ is the alphabet composed of disjoint sets of nonterminals, N, and terminals, Δ,
- R is a set of production rules. N contains the *start symbol*, denoted by S [1].

Unrestricted grammars are the general family of grammars – they properly include context-sensitive, context-free, and regular grammars. Grammars of this family can be recognized by a Turing machine or two-pushdown automaton. Neither side of rules belonging to these grammars is restricted in any way – it can consist of any string of nonterminal and terminal symbols.

The family of languages generated by unrestricted grammars is closed under the operations of intersection, union, concatenation and iteration, but not under set difference.

Unrestricted grammars generally use Kuroda normal form [3], which normalizes all grammar rules into one of the following forms:

- $A \to BC$,
- $AB \to DC$,
- $A \to a$,
- $A \to \varepsilon$.

This form is similar to Chomsky normal form for context-free languages – it adds the context-sensitiveness of $AB \to DC$. If $A = D$, the grammar is also in Penttonen normal form, a special case of Kuroda normal form.

Any grammar can be algorithmically converted into an equivalent grammar in normal form [2] – the algorithm works as follows. It modifies the original grammar, and after each made change, it moves the set of production rules that adhere to the normal form to the new grammar. These changes are made as follows:

1. In all rules, replace every terminal by a nonterminal and include these productions in the new grammar.
2. In all rules whose left-hand side is longer than the right-hand side, extend the left-hand side by a nonterminal that finally derives into ε.
3. For each rule, in which a nonterminal derives into a different nonterminal, add a rule whose right-hand side is composed of the derived nonterminal followed by the left-hand side nonterminal of every rule the derived nonterminal is used in.
4. For every context-free production rule whose right-hand side is longer than two, split it into several connecting rules.
5. For every context-sensitive production rule, pick the first pair of nonterminals on the left-hand side of the rule and ensure that the pair derives into the first nonterminal on the right-hand side of the rule followed by a supplementary nonterminal.
   Subsequently, add a rule whose left-hand side consists of the supplementary nonterminal and the rest of the original rule. Right-hand side of this rule consists of the remaining nonterminals of the right-hand side of the original rule. In case the right-hand side of the original rule was of a length greater than three, further modify the rule until it conforms to the normal form.

3. Algorithm Description

Given a grammar, $G = ({}_G\Sigma, {}_GR)$, in Penttonen normal form, and a string $w = a_1a_2\ldots a_n$ with $a_i \in {}_G\Delta$, $1 \leq i \leq n$, for some $n \geq 1$, this algorithm decides, whether w is a sentence of $L(G)$ in a bottom-up way. For its work, it uses a matrix of sets, CV – *CurrentVersion*, where $CV[i,j]$, $1 \leq i \leq j \leq n$. Each such version represents a version of the syntax tree. Other than this, the algorithm uses several sets that hold additional information – the most important one being V – *Versions*, whose members are versions of the CV matrix the algorithm has generated so far. Therefore, it contains different versions of the derivation tree, of which all are mutually exclusive.

Every member of $CV[i,j]$ has its own BLQ set – every *BlackListQueue* contains coordinates of all cells of the corresponding matrix that contain context-sensitive nonterminals, but whose ignored coordinates have not been set yet. As mentioned above, every cell of the CV matrix contains a set of nonterminals that are likely in this phase of the derivation process. For every such nonterminal A, there exists a *Predecessor* set P_A, containing references to all of its predecessors. Any nonterminal can be marked as context-sensitive, which indicates that it originated from a context-sensitive rule.

The algorithm starts the parsing process by scanning the input string, adding A to $CV[i,i]$, if A $\to a_i \in {}_GR$. For every such nonterminal, an empty set of its predecessors is constructed. Afterwards, the algorithm iterates through the matrix and constructs the sets $CV[i,j]$. For every pair $CV[i,j]$ and $CV[j+1,k]$, it checks whether $CV[i,j]$ is a member of BLQ, which indicates that the set contains context-sensitive nonter-

minals, but its ignored coordinates have not been appointed yet. These are the coordinates of the first right neighbour $CV[i,j]$ is compared with, whose nonterminals must not be used in combination with the context-sensitive nonterminals of the examined set – this is to ensure the consistency of the derivation tree. If the ignored coordinates have not been set yet, those of $CV[j+1,k]$ are used, and $CV[i,j]$ is removed from BLQ.

The behaviour of the following step is dependent on whether the ignored coordinates of $CV[i,j]$ are equal to $CV[j+1,k]$. If they are unequal, the algorithm proceeds as follows. Any nonterminal B that satisfies A $\in CV[i,j]$, C $\in CV[j+1,k]$, AB $\rightarrow$ AC $\in {}_GR$, is added to $CV[j+1,k]$. This implies that C $\Rightarrow^*$ $a_{j+1}\ldots$ a_k, and therefore B $\Rightarrow^*$ $a_{j+1}\ldots a_k$. An empty set P_B, used to refer to predecessors of the nonterminal, is constructed. A reference to nonterminal B is then added to sets P_A and P_C, in case either of the original nonterminals is to be deleted in the event of version splitting. This process is repeated until $CV[j+1,k]$ cannot be extended anymore. If any context-sensitive nonterminals were added and the set's ignored coordinates have not been set yet, a reference to this set is added to BLQ.

Any nonterminal A is then added to the set $CV[i,k]$, if it satisfies B $\in CV[i,j]$, C $\in CV[j+1,k]$, A $\rightarrow$ BC $\in {}_GR$, because B $\Rightarrow^*$ $a_i\ldots a_j$, C $\Rightarrow^*$ $a_{j+1}\ldots a_k$, and therefore A $\Rightarrow^*$ $a_i\ldots a_k$. An empty set of predecessors, P_A, is created for each added nonterminal A. A reference to nonterminal A is added to sets P_B and P_C.

If the ignored coordinates are equal to $CV[j+1,k]$, a copy of the matrix is created before the first matching rule is applied. All context-senstive nonterminals of $CV[i,j]$, including their predecessors, are deleted from the copy using predecessor sets, P_A for a nonterminal A, to recursively detect all of a nonterminal's predecessors. All of the subsequently added nonterminals are then saved to the copy instead of CV. After the scan is completed, this copy is added to V.

Once no set can be extended, it is examined whether S $\in CV[1,n]$, so S $\Rightarrow^*$ $a_1\ldots a_n$. If the check is passed, the algorithm announces **ACCEPT**. Otherwise, CV is removed from V, a member is appointed as the new CV, and parsing is continued for this version of the matrix. If V is empty, and therefore no new CV can be appointed, there are no alternative derivation trees to be constructed. In such case, the algorithm announces **REJECT**.

The main part of the algorithm, which describes nonterminal set matrix traversal, is described in Algorithm 1. This part also describes the final testing for presence of the starting nonterminal.

Algorithm 1 Matrix traversal

Input:
a grammar, G = $({}_GE, {}_GR)$, in Penttonen normal form; $w = a_1a_2\ldots a_n$ with $a_i \in {}_G\Delta$, $1 \leq i \leq n$, for some $n \geq 1$.

Output:
ACCEPT if $w \in L(G)$;
REJECT if $w \notin L(G)$.

```
1:  introduce set V = ∅;
2:  introduce matrix of sets CV, where CV[i,j] = ∅
    for 1 ≤ i ≤ j ≤ n, and add CV to V;
3:  introduce set BLQ = ∅;
4:  for i = 1 to n do
5:      if A → a_i ∈ _GR then
6:          add A to CV[i,i];
7:          introduce set P_A = ∅ holding references to
             A's predecessors;
8:  parse_loop:
9:  repeat
10:     for level = 1 to n − 1 do
11:         for i = 1 to n − level do
12:             set k to i + level;
13:             for offset = 0 to level − 1 do
14:                 set j to i + offset;
15:                 ApplyRules(i, j, k);
16: until no change;
17: if S ∈ CV[1,n] then
18:     ACCEPT
19: else
20:     remove CV from V;
21:     if V ≠ ∅ then
22:         pick an element of V and set it as CV;
23:     else REJECT;
24: goto parse_loop;
```

The indices acquired in Algorithm 1 are used to locate sets representing substrings of the input string, and to subsequently appoint coordinates of the nonterminal that joins these substrings. The process is further described by Algorithm 2.

Because the grammar, G, is finite, the algorithm always finishes. The proof of soundness is analogical to that of the original CYK algorithm [1] (p. 120 – 121).

Time complexity of the algorithm depends on both size of the grammar, |G|, and length of the input string, n. In the worst case scenario, each iteration of the outermost loop reduces a single rule that triggers version splitting. In this case, the complexity reaches

Algorithm 2 Rule Application

procedure APPLYRULES(i, j, k)
  if $CV[i,j] \in BLQ$ **then**
    set its ignored coordinates to $[j+1,k]$;
    remove $CV[i,j]$ from BLQ;
  if A $\in CV[i,j]$, C $\in CV[j+1,k]$, rhs(p) = AC for some A, C $\in {}_GN$, p $\in {}_GR$ **then**
    if coordinates ignored by $CV[i,j]$ are unequal to $[j+1,k]$ **then**
      if AB $\rightarrow$ AC $\in {}_GR$ for some B $\in {}_GN$ **then**
        add B to $CV[j+1,k]$ and mark it as context-sensitive;
        if ignored coordinates of $CV[j+1, k]$ are not set **then**
          add $CV[j+1,k]$ to BLQ;
      if B $\rightarrow$ AC for some B $\in {}_GN$ **then**
        add B to $CV[i,k]$;
      introduce set $P_B = \varnothing$ holding references to B's predecessors;
      add a reference to the nonterminal B to sets P_A and P_C;
    else
      if this instance of A is context-sensitive **then continue to next nonterminal;**
      create a copy of CV, its BLQ, and all sets of predecessors;
      remove all predecessors of context-sensitive nonterminals in $copy[i,j]$;
      remove all context-sensitive nonterminals from $copy[i,j]$;
      if AB $\rightarrow$ AC $\in {}_GR$ for some B $\in {}_GN$ **then**
        add B to $copy[j+1,k]$ and mark it as context-sensitive;
        if ignored coordinates of $copy$ $[j+1,k]$ are not set **then**
          add $copy[j+1,k]$ to the copy's BLQ;
      if B $\rightarrow$ AC $\in {}_GR$ for some B $\in {}_GN$ **then**
        add B to $copy[i,k]$;
      introduce set $P_B = \varnothing$ holding references to B's predecessors;
      add a reference to the nonterminal B to sets P_A and P_C;
      add the copy to V;

$O(n^3 \cdot |G|^3)$, as described by Equation 1.

In that case, the algorithm creates a total of |G| additional matrices, of which each cell can hold |G| reduced nonterminals. That also means the space complexity is $O(n^2 \cdot |G|^3)$, as described by Equation 2.

$$t = (|G|+1)^2 \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} \sum_{k=0}^{i-1} |G|$$
$$t = \frac{n(n^2-1)\ |G|(|G|+1)^2}{6} \tag{1}$$
$$O(t) = O(n^3 \cdot |G|^3)$$

$$s = (|G|+1) \sum_{i=1}^{n} |G| i$$
$$s = \frac{(|G|+1)(n-|G|+1)(|G|+n)}{2} \tag{2}$$
$$O(s) = O(n^2 \cdot |G|^3)$$

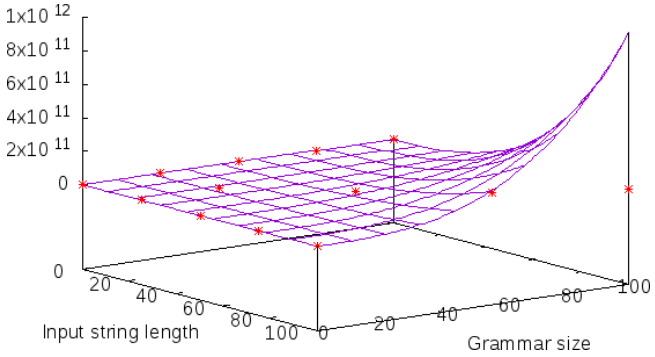
The experimental results support these assumptions, as they show a steady growth when increasing one of the parameters, and an exponential growth when values of both are high. The values listed in Table 1 are the average of a total of three runs with the specified parameters. The data was acquired using Memcheck and Callgrind utilities available as a part of the `Valgrind` memory checker suite. This data is further visualized and compared to the analytical values in Figure 1.

Table 1. Time and space complexity test results for a total of 42 input grammars and strings with an overall success rate of ~40.5 %.

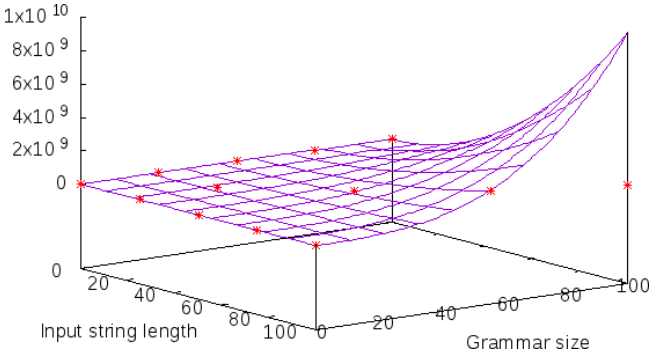
n	|G|	Instructions	Memory [B]
0	0	85100	2270000
0	25	99933	2780000
0	50	109000	3226667
0	75	119067	3750000
0	100	128267	4210000
25	0	693350	43700000
25	25	1900000	165566667
50	0	2500000	220800000
50	50	45866667	3532333333
75	0	4850000	553566667
75	75	294900000	21086666667
100	0	9066667	1218000000
100	100	847666667	60819000000

4. Prototype Implementation

The working prototype, which implements the algorithm, is programmed in C++. The language was chosen because it allows object-oriented programming, as well as its standard containers and operations available on them.



(a) Time complexity of the presented algorithm estimated at $O(n^3 \cdot |G|^3)$.



(b) Space complexity of algorithm reaches $O(n^2 \cdot |G|^3)$ in worst case scenario.

Figure 1. Time and space of the presented algorithm both reach at least cubic values because of the context-sensitive nature of the discussed topic. The purple grids represent analytical values of estimated complexities, while the red points represent average experimental evalues from Table 1.

The program is divided into several classes, of which each has a distinct role in the process of parsing user settings, versioning, and parsing the input string to finally decide membership of the input string:

Grammar Adapter This module parses rules of a user grammar according to the predefined format. Before saving a rule, it checks if the rule adheres to restrictions of Penttonen normal form. Processed rules are subsequently added to `Grammar`, as described below.

Input String Adapter This module works in a way similar to the previous one. It scans the input file and converts it into a `vector` of terminals that are later parsed according to `Grammar`. The tokens are divided by a white space, therefore the input string can be of unlimited length.

Grammar Structure This module stores grammar rules used to detect derivations. It consists of three separate lists of rules, of which each set is used during a different phase of the parsing process – *terminal rules*, which are used to construct the starting sets of nonterminals, *context-free rules* and *context-sensitive rules*, which are used during the later phases of the process. This module is responsible for verifying right-hand side match of the appropriate rule during the parsing, as well as retrieving the left-hand side nonterminals in case of a positive match.

Versioning System This module is responsible for storing and managing viable versions of the nonterminal matrix the program has generated so far. Each version of the matrix has a corresponding *blacklist queue* of context-sensitive sets, as described in Section 3. The module keeps a reference to the *current version* of the matrix, which is modified during the application of production rules. This module announces the final rejection of the input string, if unable to appoint a new version of the parsing matrix for processing.

Parsing Matrix This class presents a two-dimensional map of matrix cells. Each such cell contains a set of nonterminals used during the parsing process, and lists of rules that have been used to generate the member nonterminals of the corresponding set. These lists are used to prevent both endless looping of context-sensitive rule application, and application of rules that would lead to adding a nonterminal that had previously been deleted from the set because of a context collision.

These modules are encapsulated by a single `Parser` object. It parses the input files, and saves the adapters' output. As described in algorithm 1, it creates the first version of the matrix, initializes it by creating the starting sets of nonterminals according to the previously obtained grammar, and sets it as *current version*. Afterwards, the second phase of the parsing process begins. Once a matrix cannot be changed any further, it checks for the presence of the starting nonterminal. In case of failure, the versioning system attempts to appoint a new *current version*.

5. Conclusions

This paper deals with the topic of context-sensitive grammars as special cases of unrestricted grammars, and applications of their normal forms. It presents an algorithm capable of parsing these grammars and a software implementation of this algorithm that implements this algorithm.

The algorithm is based on Cocke-Younger-Kasami parsing algorithm and therefore works in a bottom-

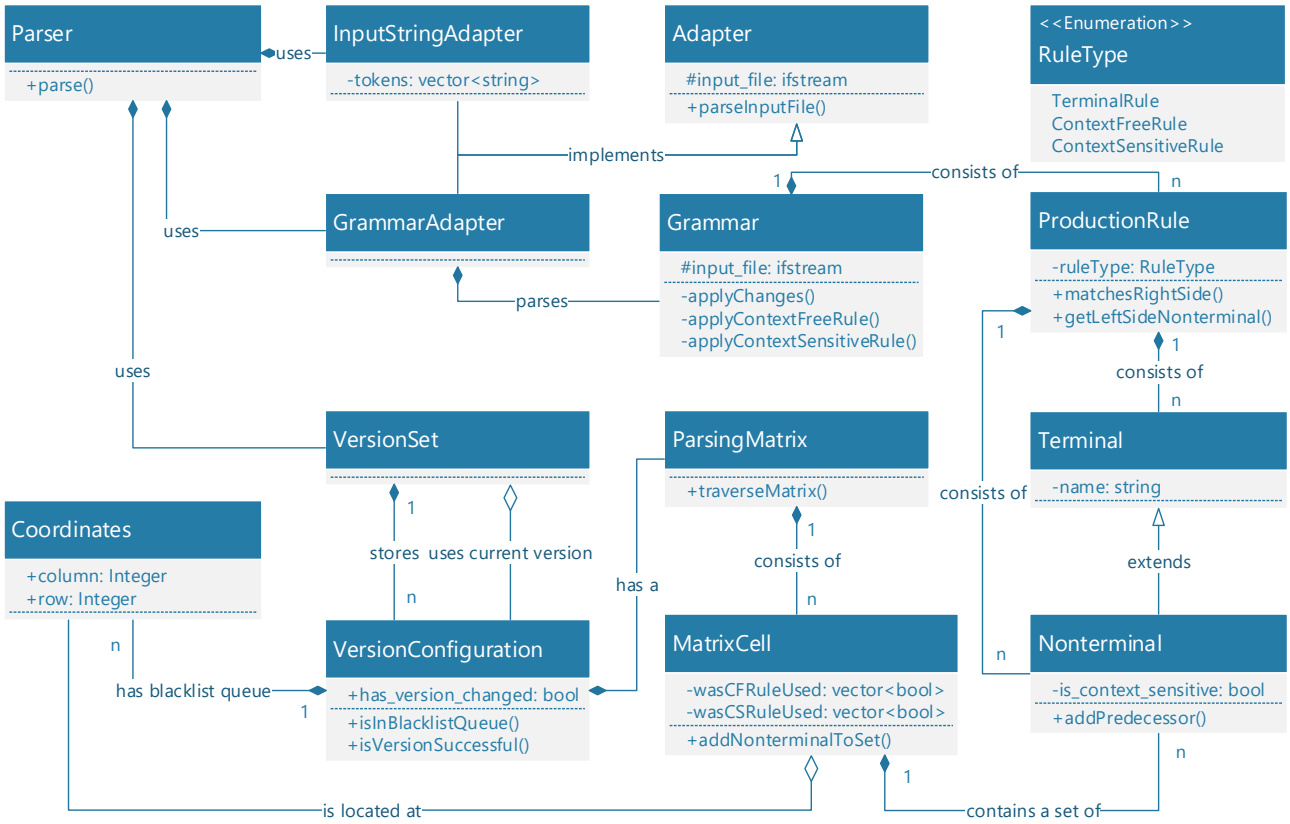


Figure 2. Diagram of classes as implemented in the working prototype. An instance of `Parser` class encapsulates the two main modules, of which each has a distinct role – `Adapter` scans the input files holding input string and grammar, and `VersionSet` stores the considered versions of the parsing matrix.

up way. It is capable of parsing any context-sensitive grammar in Penttonen normal form, including ambiguous grammars.

As this work focuses on a topic that has not been thoroughly researched yet, it aims to serve mainly as a base for subsequent research. In the future, I would like to expand the algorithm to accept grammars in additional normal forms, which would allow for more extensive application.

Acknowledgements

I would like to thank my supervisor Prof. RNDr. Alexander Meduna, CSc. for his help.

References

[1] MEDUNA, Alexander, 2008. *Elements of compiler design*. Boca Raton: Auerbach Publications, xiii, 286 p. ISBN 1420063235.

[2] MEDUNA, Alexander, 2000. *Automata and languages: theory and applications*. 2000 edition. London: Springer, xv, 916 p. ISBN 1852330740.

[3] ROZENBERG, Grzegorz and Arto SALOMAA, 1997. *Handbook of formal languages: word, language, grammar*. Berlin: Springer, xvii, 873 pages; 23 cm. ISBN 9783642638633.