An Alternative Formulation of Cocke–Younger–Kasami's Algorithm*

Peter R.J. Asveld

Department of Computer Science, Twente University of Technology
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract – We provide a reformulation of Cocke–Younger–Kasami's algorithm for recognizing context-free languages in which there are no references either to indices of table entries or to the length of the input string. Some top-down analogues of this functional approach are discussed as well.

Keywords: context-free grammar, normal form, recognition, parsing

---

Let $G = (V, \Sigma, P, S)$ be a context-free grammar with alphabet V, terminal alphabet Σ, set of productions P and start symbol S. The set of nonterminal symbols will be denoted by N, i.e. $N = V - \Sigma$. For each set X, $\mathcal{P}(X)$ denotes the power set of X.

Cocke–Younger–Kasami's algorithm —or CYK-algorithm for short— is usually presented as follows.

Algorithm 1. Let $G = (V, \Sigma, P, S)$ be a context-free grammar in Chomsky normal form (without the rule $S \to \lambda$) and let $a_1 a_2 \cdots a_n$ $(n \geq 1)$ be a string with $a_k \in \Sigma$ $(1 \leq k \leq n)$. Form the strictly upper-triangular $(n+1)\times(n+1)$ recognition matrix T as follows, where each element t_{ij} is a subset of $N = V - \Sigma$ and is initially empty.

begin
  for $i := 0$ **to** $n-1$ **do**
    $t_{i,\,i+1} := \{A \mid A \to a_{i+1} \in P\}$
  for $d := 2$ **to** n **do**
  for $i := 0$ **to** $n-d$ **do**
    begin $j := d + i$;
      $t_{ij} := \{A \mid \exists k\ (i+1 \leq k \leq j-1)\colon \exists B\ (B \in t_{ik})\colon \exists C\ (C \in t_{kj})\colon A \to BC \in P\}$
    end
end.

Then $a_1 a_2 \cdots a_n \in L(G)$ if and only if $S \in t_{1n}$. □

The above formulation is adapted from [4], but similar versions can be found in other text books like [1] or [5]. A striking feature of these formulations is the reference to the numbers i, j and k and to the length n of the input string. These numbers refer to an explicitly mentioned implementation —viz. the matrix T— rather than the essence of the algorithm. Now the obvious question is whether we can get rid of those numbers. We will answer this question in the affirmative.

To this end we define functions $f : \Sigma^+ \to \mathcal{P}(N^+)$ and $g : \mathcal{P}(N^+) \to \mathcal{P}(N)$ by:

- For each nonempty word w over Σ we define f as the length-preserving finite substitution generated by

---

* This note appeared in *Bull. Eur. Assoc. for Theor. Comp. Sci.* (1994) No. 53, 213–216.

$$f(a) = \{A \mid A \to a \in P\}$$

and extended by

$$f(w) = f(a_1)f(a_2)\cdots f(a_n) \quad \text{if} \quad w = a_1a_2\cdots a_n \quad (a_k \in \Sigma,\ 1 \le k \le n).$$

- For each ω in N^+ we define

$$g(\omega) \ = \ \bigcup \{\, g(\phi) \otimes g(\psi) \mid \phi, \psi \in N^+,\ \ \omega = \phi\psi \,\} \tag{1}$$

where for X and Y in $\mathcal{P}(N)$ the binary operation $\otimes$ is defined by

$$X \otimes Y \ = \ \{A \mid A \to BC \in P, \ \text{with} \ B \in X \ \text{and} \ C \in Y\}.$$

- For each language M over N, $g(M)$ is defined by

$$g(M) \ = \ \bigcup \{\, g(\omega) \mid \omega \in M \,\}.$$

The CYK-algorithm can now be formulated as

Algorithm 2. Let $G = (V, \Sigma, P, S)$ be a context-free grammar in Chomsky normal form (without the rule $S \to \lambda$) and let w be a string over Σ. Compute $g(f(w))$ and determine whether S belongs to $g(f(w))$.

Clearly, we have $w \in L(G)$ if and only if $S \in g(f(w))$. □

Since the binary operation $\otimes$ is associative, (1) may be considered as a matrix product and an implementation using a recognition matrix is an obvious choice. Similarly, (1) also suggests implementations based on dynamic programming, an upper triangular matrix of parallel processors, or a systolic approach; cf. [8,9].

The CYK-algorithm can be generalized for arbitrary λ-free context-free grammars rather than for λ-free grammars in Chomsky normal form; cf. [2] for details. Algorithm 2 can be modified accordingly, but the price we have to pay is that we lose the simplicity of the operator $\otimes$.

The CYK-algorithm is a bottom-up algorithm for recognizing λ-free context-free languages. Can we also proceed in a similar top-down fashion? Yes, as in Algorithm 3 for which we need the following

Definition. Let $G = (V, \Sigma, P, S)$ be a context-free grammar and $N = V - \Sigma$. The set $T(\Sigma, N)$ of *terms* over (Σ, N) is the smallest set satisfying

(i) λ is a term in $T(\Sigma, N)$ and each a ($a \in \Sigma$) is a term in $T(\Sigma, N)$.

(ii) For each A in N and each term t in $T(\Sigma, N)$, $A(t)$ is a term in $T(\Sigma, N)$.

(iii) If t_1 and t_2 are terms in $T(\Sigma, N)$, then their concatenation $t_1 t_2$ is also a term in $T(\Sigma, N)$. □

Note that for any two sets of terms S_1 and S_2 ($S_1, S_2 \subseteq T(\Sigma, N)$) the entity $S_1 S_2$, defined by $S_1 S_2 = \{t_1 t_2 \mid t_1 \in S_1,\ t_2 \in S_2\}$, is also a set of terms over (Σ, N).

Algorithm 3. Let $G = (V, \Sigma, P, S)$ be a context-free grammar in Chomsky normal form (without the rule $S \to \lambda$) and let w be a string over Σ.

Each nonterminal symbol A in N is considered as a function from $\Sigma^* \cup \{\perp\}$ to $\mathcal{P}(T(\Sigma, N))$ defined as follows. (The symbol $\perp$ will be used to denote "undefined".) First, $A(\perp) = \varnothing$ and $A(\lambda) = \{\lambda\}$ for each A in N. If the argument x of A is a word of length 1 —i.e. x equals a for some a in Σ— then

$$A(a) \ = \ \{\lambda \mid A \to a \in P\}$$

and in case the length $|x|$ of the word x is 2 or more, then

$$A(x) = \bigcup \{B(y)C(z) \mid A \to BC \in P, \quad y, z \in \Sigma^+, \quad x = yz\}. \tag{2}$$

Finally, we compute $S(w)$ and determine whether λ belongs to $S(w)$.

It is straightforward to show that $w \in L(G)$ if and only if $\lambda \in S(w)$. □

Algorithm 3 is a simple recursive descent recognition algorithm that can be implemented in many ways; cf. e.g. the divide-and-conquer approach in [3]. Since the "calls" of $B(y)$ and $C(z)$ in (2) are mutually independent a parallel implementation (e.g. on a parallel random access machine [10]) is a suitable choice. Due to the fact that G does not contain λ-productions the total number of recursive calls during the computation of $S(w)$ is at least $2 \cdot |w| - 1$.

Apart from efficiency gained by an appropriate implementation we can improve upon Algorithm 3 by starting from Greibach 2-form rather than Chomsky normal form. Remember that a λ-free context-free grammar $G = (V, \Sigma, P, S)$ is in *Greibach 2-form* if the productions are of one of the following forms: $A \to aBC$, $A \to aB$ and $A \to a$ with $a \in \Sigma$ and $A, B, C \in N$; cf. [4].

Algorithm 4. Let $G = (V, \Sigma, P, S)$ be a λ-free context-free grammar in Greibach 2-form and let w be a string over Σ. The algorithm is as the previous one except that (2) is replaced by

$$\begin{aligned} A(x) = {} & \bigcup \{B(y)C(z) \mid A \to aBC \in P, \ y, z \in \Sigma^+, \ x = ayz\} \cup \\ & \bigcup \{B(y) \mid A \to aB \in P, \ y \in \Sigma^+, \ x = ay\}. \end{aligned} \tag{3}$$

Still we have that $w \in L(G)$ if and only if $\lambda \in S(w)$. □

In any implementation of this algorithm the number of recursive calls in computing $S(w)$ is at least $|w|$.

Example. Let $\#_\sigma(w)$ denote the number of times the symbol σ occurs in the word w. Consider the language $L_0 = \{w \in \{a, b\}^+ \mid \#_a(w) = \#_b(w)\}$ which is generated by the following λ-free grammar in Greibach 2-form.

$$\begin{aligned} S &\to aSB \mid aBS \mid bSA \mid bAS \mid aB \mid bA \\ A &\to aS \mid a \\ B &\to bS \mid b \end{aligned}$$

Applying Algorithm 4 yields

$$\begin{aligned} S(x) = {} & \bigcup \{S(y)B(z) \mid y, z \in \Sigma^+, \ x = ayz\} \cup \\ & \bigcup \{B(y)S(z) \mid y, z \in \Sigma^+, \ x = ayz\} \cup \\ & \bigcup \{S(y)A(z) \mid y, z \in \Sigma^+, \ x = byz\} \cup \\ & \bigcup \{A(y)S(z) \mid y, z \in \Sigma^+, \ x = byz\} \cup B(a \,]\, x) \cup A(b \,]\, x), \\ A(x) = {} & S(a \,]\, x), \\ B(x) = {} & S(b \,]\, x), \end{aligned}$$

where $u \,]\, v = w$ if $v = uw$, and $\perp$ otherwise ($u, v, w \in \Sigma^*$). Similarly, we define $u \,[\, v = w$ if $u = wv$, and $\perp$ otherwise. Remember that for each nonterminal symbol A, we have $A(\perp) = \varnothing$.

These three equalities reduce to

$$\begin{aligned} S(x) \;=\; & \bigcup \{S(a]\,x\,[b),\ S(ab]\,x),\ S(b]\,x\,[a),\ S(ba]\,x)\} \cup \\ & \bigcup \{S(y)S(b]\,z),\ S(b]\,y)S(z) \mid y, z \in \Sigma^+,\ x = ayz\} \cup \\ & \bigcup \{S(y)S(a]\,z),\ S(a]\,y)S(z) \mid y, z \in \Sigma^+,\ x = byz\}. \end{aligned}$$

As examples consider $S(abba) = S(ba) \cup S(b)S(a) = \{\lambda\} \cup S(b)S(a)$, and $S(aba) = S(a) \cup S(a)S(\lambda) = S(a)$. Since $\lambda \in S(abba)$ and $\lambda \notin S(aba)$, we have $abba \in L_0$ and $aba \notin L_0$, respectively. □

Instead of Greibach 2-form we may also take other normal forms as starting point in order to obtain more efficient algorithms. Other possibilities, for instance, are the double Greibach 2-form and the supernormal form of [7]. However, in those cases (3) becomes much more complicated.

Removing details referring to possible implementations or data structures clearly yields more functional descriptions of recognition algorithms. (For a functional variant of Earley's algorithm we refer to [6].) Although some of our algorithms are rather inefficient, they may serve as a basis for a general approach to recognition and parsing algorithms, a subject that is one of the main topics in [11].

Acknowledgements. I am indebted to Rieks op den Akker and Klaas Sikkel for some critical remarks.

References

1. A.V. Aho & J.D. Ullman: *The Theory of Parsing, Translation and Compiling – Volume I: Parsing* (1972), Prentice-Hall, Englewood Cliffs, NJ.
2. H.J.A. op den Akker: Recognition methods for context-free languages (1991), Memoranda Informatica 91-30, Dept. of Comp. Sci., University of Twente, Enschede, the Netherlands.
3. A. Bossi, N. Cocco & L. Colussi: A divide-and-conquer approach to general context-free parsing, *Inform. Process. Lett.* **16** (1983) 203-208.
4. M.A. Harrison: *Introduction to Formal Language Theory* (1978), Addison-Wesley, Reading, Mass.
5. J.E. Hopcroft & J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (1979), Addison-Wesley, Reading, Mass.
6. R. Leermakers, A recursive ascent Earley parser, *Inform. Process. Lett.* **41** (1992) 87-91.
7. H.A. Maurer, A. Salomaa & D. Wood: A supernormal-form theorem for context-free grammars, *J. Assoc. Comp. Mach.* **30** (1983) 95-102.
8. A. Nijholt: Overview of parallel parsing strategies, Chapter 14 in M. Tomita (ed.): *Current Issues in Parsing Technology* (1991), Kluwer, Boston.
9. A. Nijholt: The CYK-approach to serial and parallel parsing, *Proc. Seoul Internat. Conf. on Natural Language Processing SICONLP'90* (1990) 144-155.
10. W.J. Savitch & M.J. Stimson: Time bounded random access machines with parallel processing, *J. Assoc. Comp. Mach.* **26** (1979) 103-118.
11. N. Sikkel: *Parsing Schemata* (1993), Ph.D. Thesis, Dept. of Comp. Sci., University of Twente, Enschede.