Membership in context free languages with CYK*

Douglas Martins[1], **Gustavo Zambonin**[1]

[1]Departamento de Informática e Estatística, Universidade Federal de Santa Catarina
88040-900, Florianópolis, Brazil

{marcelino.douglas,gustavo.zambonin}@posgrad.ufsc.br

1. Introduction

A formal language can be used as an important instrument to represent syntactic characteristics of logical and mathematical constructs, thus allowing problems from these areas to be expressed differently and solved through an algorithm. For instance, given a piece of code, we wish to know if it is correctly written, according to the rules of its underlying programming language. An algorithm that solves this problem may use a formalism known as a grammar to represent the source code.

A grammar generates every word in a language by means of production rules, that is, a set of transformations. Grammars can be classified, according to the Chomsky hierarchy [Chomsky 1959], in four types: regular, context-free, context-sensitive and unrestricted. This classification is directly related to the classes of problems that languages are able to solve, *i.e.* unrestricted grammars can solve the most complex problems that are still computable. Furthermore, note that every problem with a Boolean answer can be expressed as a membership problem in a language (the set of words that represent positive solutions). Algorithms that solve these problems in the context of grammars are called parsers.

As discussed above, syntax of programming languages is usually described in the form of a context-free grammar. This happens because regular languages cannot deal with common source code idioms, such as the presence of balanced parenthesis. Even though this description is concise, the act of parsing strings may be computationally complex. Hence, we make use of strategies such as the conversion of context-free grammars into an equivalent grammar in Chomsky normal form (CNF), that allows easier parsing of strings through an even simpler description of the grammar.

This is exactly the situation presented in [Guimarães 2007]. Context-free grammars in CNF are given, and a parser must be written to solve the problem. We choose the CYK algorithm, present it in Section 2 and discuss the solution programmed in Section 3.

2. Cocke–Younger–Kasami algorithm

The CYK algorithm was discovered independently by Cocke [Cocke and Schwartz 1970], Younger [Younger 1967] and Kasami [Kasami 1966]. It is a bottom-up parser that uses dynamic programming to decide whether a word is member of a context-free language. Consider a grammar as a 4-tuple $G = (V, \Sigma, R, S)$, where V is a finite set of variables, Σ is a finite set of terminals such that $V \cap \Sigma = \emptyset$, $S \in V$ is the start variable, and $\forall A, B, C \in V, \forall \alpha \in \Sigma$, R is a finite set of rules with the forms $A \to \alpha$ or $A \to BC$. This definition of a CNF grammar is due to Sipser [Sipser 2006], and note that converting

*As submitted to the INE410113 class (Theory of Computation). Source code.

any context-free grammar to its equivalent CNF normal form is possible [Sipser 2006, Theorem 2.9]. We present a description of CYK and discuss it below.

The main notion of CYK is the fact that every word in the language can be subdivided into a prefix and a suffix, starting with the base case, when there are only terminals, exhibiting its bottom-up nature. The CNF restriction for rules allows exactly this splitting behavior. Dynamic programming is employed to solve smaller problems, *i.e.* check if substrings of an input are in the language.

Algorithm 1 CYK parser

Input: $w \in \Sigma^*, G$ ▷ word, grammar in CNF
Output: $v \in \{\mathbf{T}, \mathbf{F}\}$ ▷ Boolean value that represents whether $w \in G$
1: $n \leftarrow |w|$ ▷ size of w
2: $d \leftarrow [\{\}_0^0, \dots, \{\}_{n-1}^0, \{\}_0^1, \dots, \{\}_{n-1}^{n-1}]$ ▷ square $n \times n$ matrix of empty sets
3: **for** $i \leftarrow 0, \dots, n-1$ **do**
4:     **if** $(A \rightarrow w[i]) \in G.R$ **then**
5:         $d_i^i \leftarrow d_i^i \cup \{A\}$ ▷ if rule produces the i-th letter of w
6:     **end if**
7: **end for**
8: **for** $\ell \leftarrow 1, \dots, n-1$ **do**
9:     **for** $r \leftarrow 0, \dots, n-\ell-1$ **do**
10:         **for** $t \leftarrow 0, \dots, \ell-1$ **do**
11:             $\mathcal{L} \leftarrow d_r^{r+t}$ ▷ rules that generate the $w[r : r+t]$ prefixes
12:             $\mathcal{R} \leftarrow d_{r+t+1}^{r+\ell}$ ▷ rules that generate the $w[r+t+1 : r+\ell]$ suffixes
13:             **for** $(B, C) \in \mathcal{L} \times \mathcal{R}$ **do**
14:                 **if** $(A \rightarrow BC) \in G.R$ **then**
15:                     $d_r^{r+\ell} \leftarrow d_r^{r+\ell} \cup \{A\}$ ▷ if rule produces $w[r : r+\ell]$
16:                 **end if**
17:             **end for**
18:         **end for**
19:     **end for**
20: **end for**
21: $v \leftarrow (S \in d_0^{n-1})$ ▷ if the starting rule is in the top right matrix cell

---

Consider Algorithm 1, that receives as input a word w and a grammar G. We refer to its rules by $G.R$. We create an upper triangular $n \times n$ matrix d, where n is the length of w, and by Lines 3–7, its diagonal is filled with all rules that produce each letter in w. Then, by Lines 8–20, for every substring length ℓ that is not trivial, create all possible prefix-suffix pairs by modifying their starting indices r, t. Compute the Cartesian product of the rules that generate these substrings, and check if there is a rule that generates one of the pairs. If so, add this rule to the whole substring set in the corresponding cell. This will generate the diagonal correspondent to substrings of size ℓ.

[Hopcroft et al. 2006, Example 7.34.] Let G be the CNF grammar below, and

$w = baaba$.

$$\begin{aligned} S &\to AB \mid BC \\ A &\to BA \mid a \\ B &\to CC \mid b \\ C &\to AB \mid a. \end{aligned}$$

The resulting matrix for the CYK algorithm is

$$d = \begin{bmatrix} \{B\} & \{S, A\} & \emptyset & \emptyset & \{S, A, C\} \\ \emptyset & \{A, C\} & \{B\} & \{B\} & \{S, A, C\} \\ \emptyset & \emptyset & \{A, C\} & \{S, C\} & \{B\} \\ \emptyset & \emptyset & \emptyset & \{B\} & \{S, A\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \{A, C\} \end{bmatrix}.$$

Since $S \in d_0^{n-1}$, then $w \in L(G)$.

The algorithm has a complexity of $\mathcal{O}(n^3 \cdot s)$, n as above and s as the quantity of variables and terminals for every $r \in R$. Intuitively, this is the case since there are n^2 cells in the matrix d, each populated with a set linear in size. Furthermore, this happens because, unlike parsers that are restricted to specific types of grammars, CYK can parse every context-free grammar. Still, it was proven by Lee [Lee 2002] that context-free parsing is equivalent to Boolean matrix multiplication, thus implying that this complexity can be turned sub-cubic.

3. Implementation

The full source code for the parser can be read in Appendix A. We briefly discuss it in the sequence. Heavy usage of unordered containers is applied, since there is no time access overhead, that is, $\mathcal{O}(1)$ element look-ups. Further, we create customized hashing functions to allow insertion of complex objects into these structures. We parse the input with regular expression rules created from the variables and terminals sets. After, there are two helper functions that return, respectively, all rules that generate a pair of symbols, and the Cartesian product of two character sets. Note that we pair any terminals with a special character $\#$ — this way, only one function needs to be created.

In Lines 115–144, the main function is defined. It is very similar to Algorithm 1, featuring only two differences in the form of optimizations: early exit if any element in the diagonal is an empty set, and memoization of rules produced by a pair of characters. With these, we need not populate the matrix if any letter of w is not produced by G, and repeated computations are prevented in the frequent case of equal character pairs. Subsequently, a wrapper function is responsible for aggregating all possible inputs of a grammar and its results, avoiding repeated computations if there are two or more equal input words. A pretty-print function is defined shortly after. Finally, in the last function, memoization of the entire grammar and its results is employed to prevent repeated cases, once again. The main obstacles were based around how to structure a grammar to allow easy iteration and comparison of rules, as well as inserting complex structures into unordered containers. Still, the program is relatively optimized, running in less than 30 ms.

References

[Chomsky 1959] Chomsky, N. (1959). On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167.

[Cocke and Schwartz 1970] Cocke, J. and Schwartz, J. (1970). *Programming languages and their compilers: Preliminary notes*. 2nd edition.

[Guimarães 2007] Guimarães, W. (2007). Sphere Online Judge ET Problem. Accessed: 21 oct 2018.

[Hopcroft et al. 2006] Hopcroft, J., Motwani, R., and Ullman, J. (2006). *Introduction to Automata Theory, Languages, and Computation*. 3rd edition.

[Kasami 1966] Kasami, T. (1966). An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages. Technical Report Coordinated Science Laboratory Report no. R-257, University of Illinois at Urbana-Champaign.

[Lee 2002] Lee, L. (2002). Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *Journal of the ACM*, 49(1):1–15.

[Sipser 2006] Sipser, M. (2006). *Introduction to Theory of Computation*. 2nd edition.

[Younger 1967] Younger, D. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

A. C++ implementation of CYK

```cpp
#include <iostream>
#include <regex>
#include <set>
#include <unordered_map>
#include <unordered_set>

using char_pair = std::pair<char, char>;
using char_set = std::set<char>;

// Provides a way to use a pair of variables as a key in unordered containers.
struct pair_hash {
  std::size_t operator()(const char_pair &p) const {
    // pretend XOR is not commutative for pairs such as 'AB' and 'BA'
    return std::hash<char>{}(p.first) ^ (std::hash<char>{}(p.second) << 1);
  }
};

// A context-free grammar is a 4-uple consisting of variables, terminals,
// productions and the start variable. Only two of these are needed here.
struct grammar {
  char start_prod;
  std::unordered_map<char, std::unordered_set<char_pair, pair_hash>> rules;
  std::vector<std::string> possible_inputs;
};

// Provides a way to use a grammar as a key in unordered containers.
struct gram_hash {
  std::size_t operator()(const grammar &g) const {
    uint32_t hash = g.start_prod;
    for (const auto &rule : g.rules) {
      hash ^= std::hash<char>{}(rule.first);
      for (const auto &pair : rule.second) {
        hash ^= rule.second.hash_function()(pair);
      }
    }
    return hash;
  }
};

// Two grammars are equal if they have the same rules and same possible inputs.
// This prevents unnecessary instantiation of CYK with two equal inputs.
struct gram_eq {
  bool operator()(const grammar &g1, const grammar &g2) const {
    return g1.rules == g2.rules && g1.possible_inputs == g2.possible_inputs;
  }
};

using input_map = std::unordered_map<std::string, bool>;
using grammar_map = std::unordered_map<grammar, input_map, gram_hash, gram_eq>;
```

```
// Creates a structure containing the description of a context-free grammar in
// Chomsky normal form, and does so through matching possible rules with
// regular expressions.
grammar load_grammar(std::istream &in) {
  grammar g{};

  try {
    std::string prod, term, line;
    in >> g.start_prod >> prod >> term;

    // end of input with multiple grammars
    if (g.start_prod == 0) {
      return g;
    }

    // various input file limitations according to original problem
    bool valid_start_prod = std::isupper(g.start_prod);
    bool valid_prod = std::all_of(prod.begin(), prod.end(), isupper);
    bool start_in_prod = prod.find(g.start_prod) != std::string::npos;
    bool no_space_term = term.find(' ') == std::string::npos;
    bool no_hashtag_term = term.find('#') == std::string::npos;
    if (!(valid_start_prod && valid_prod && start_in_prod && no_space_term &&
          no_hashtag_term)) {
      throw std::invalid_argument("input is not valid");
    }

    // ignore rest of line
    in.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    const std::regex valid_rules("[" + prod + "] -> ([" + term + "]|[" + prod +
                                 "]{2})");

    while (getline(in, line) && line != "# -> #" &&
           std::regex_match(line, valid_rules)) {
      // if rule produces only a terminal, fill the remaining pair element with
      // an useless symbol
      g.rules[line[0]].insert(
          char_pair(line[5], (line.size() == 6) ? '#' : line[6]));
    }

    constexpr uint8_t max_word_len = 50;
    while (getline(in, line) && line != "#") {
      // input must not feature letters that symbolize productions
      bool letter_is_prod = std::any_of(line.begin(), line.end(), [=](char c) {
        return prod.find(c) != std::string::npos;
      });
      if (line.size() > max_word_len || letter_is_prod) {
        throw std::invalid_argument("word is not valid");
      }
      g.possible_inputs.emplace_back(line);
    }
  } catch (const std::exception &e) {
    exit(EXIT_FAILURE);
  }

  return g;
}

// Identify which rules produce a given _ordered_ pair of symbols.
char_set get_rules_for_symbols(const grammar &g, const char_pair &p) {
  char_set result;
  for (const auto &rule : g.rules) {
    for (const char_pair &poss : rule.second) {
      if (poss == p) {
        result.insert(rule.first);
      }
    }
  }
  return result;
}

// Creates the cartesian product of two sets of characters.
std::set<char_pair> cartesian_prod(const char_set &a, const char_set &b) {
  std::set<char_pair> result;
  for (const char &a1 : a) {
    for (const char &b1 : b) {
      result.insert(std::make_pair(a1, b1));
    }
  }
  return result;
}

// CYK employs dynamic programming to test membership of a string in a
// context-free language, here represented by a grammar. It builds a triangular
// matrix of rules that produce substrings of the given input.
bool cyk(const grammar &g, const std::string &w) {
  const std::string::size_type n = w.size();
  std::vector<std::vector<char_set>> table(n, std::vector<char_set>(n));
  std::unordered_map<char_pair, char_set, pair_hash> memo;

  // fill table's diagonal with the rules that produce characters in the input
  for (uint32_t i = 0; i < n; ++i) {
    char_set rules = get_rules_for_symbols(g, char_pair(w[i], '#'));
    // if no rules produce any of the letters in the word, then it cannot be a
```

```cpp
    // member of the language
    if (rules.empty()) {
      return false;
    }
    table[i][i].insert(rules.begin(), rules.end());
  }

  // non-trivial substring lengths not featured above
  for (uint32_t l = 1; l < n; l++) {
    // starting index for a substring of length l
    for (uint32_t r = 0; r < n - l; r++) {
      // every character of the substring w[r:r+l]
      for (uint32_t t = 0; t < l; t++) {
        // the substring can be split in two parts, each produced by their
        // own rules; all rules from both parts must be considered to figure
        // out if any produces the substring, hence the cartesian product
        std::set<char_pair> prod =
            cartesian_prod(table[r][r + t], table[r + t + 1][r + l]);
        for (const char_pair &pair : prod) {
          // memoize rules producing a given pair of characters
          if (memo.find(pair) == memo.end()) {
            memo[pair] = get_rules_for_symbols(g, pair);
          }
          table[r][r + l].insert(memo[pair].begin(), memo[pair].end());
        }
      }
    }
  }

  return table[0][n - 1].find(g.start_prod) != table[0][n - 1].end();
}

// Executes CYK for all words in the list of possible inputs. Features an
// example of memoization since repeated words can appear in the input file.
input_map cyk_wrapper(const grammar &g) {
  input_map memo;
  for (const std::string &word : g.possible_inputs) {
    if (memo.find(word) == memo.end()) {
      memo[word] = cyk(g, word);
    }
  }
  return memo;
}

// Prints output according to SPOJ guidelines.
void cyk_printer(const grammar &g, input_map results, const uint32_t index) {
  std::cout << "Instancia " << index << std::endl;
  for (const std::string &word : g.possible_inputs) {
    std::cout << word;
    if (!results[word]) {
      std::cout << " nao";
    }
    std::cout << " e uma palavra valida" << std::endl;
  }
  std::cout << std::endl;
}

int32_t main() {
  grammar g{};
  grammar_map memo;
  uint32_t i = 0;

  do {
    g = load_grammar(std::cin);
    if (g.start_prod != 0) {
      if (memo.find(g) == memo.end()) {
        memo[g] = cyk_wrapper(g);
      }
      cyk_printer(g, memo[g], ++i);
    }
  } while (g.start_prod != 0);

  exit(EXIT_SUCCESS);
}
```