**Universidad de Ingeniería y Tecnología**

Ciencia de la Computación

# Natural Language Processing with Context Free Grammar

*Proyecto - Teoría de la Computación*

Autores:
Christian Ledgard Ferrero
Anthony Aguilar
Alexander Baldeon

Mayo 2020

# Índice

# 1. Introducción

Hoy en día la tecnología está presente en muchos ámbitos de nuestra sociedad. Desde vehículos que se manejan por sí solos hasta en nuestros celulares que nos permiten estar más comunicados que nunca. Unos de los avances más significativos en los últimos tiempos son los asistentes virtuales como Siri, Alexa, Google Assistant, entre otros. ¿Alguna vez se han preguntado cómo estos sistemas organizan y analizan nuestras oraciones para ponerlas en un determinado contexto? Una posible implementación es utilizar **gramáticas independientes del contexto** para organizar y luego generar un árbol de *"parseo"* con la oración. Esto se logra al tener un léxico y una gramática previamente definida.

# 2. Alcance

En nuestro proyecto nosotros realizaremos una investigación aplicada en donde desarrollaremos un algoritmo simple para realizar un parser de acuerdo a un léxico y gramática previamente definida. Utilizaremos diversas fuentes (1), (2), (3), (4), (5), (6), (7), (8) y (9).

# 3. Definición del Problema

Cómo dividir una oración, utilizando gramáticas independientes del contexto, para que el computador pueda interpretarlas correctamente. ¿Casos de ambigüedad? ¿Qué hacer en determinados casos?

# 4. Estado del Arte

El inicio del procesamiento de lenguaje natural suele remontarse a 1950 cuando Alan Turing publicó el articulo 'Computing Machinery and Intelligence' donde propone lo que conocemos como el test de Turing, un criterio de inteligencia.
Una de las primeras aplicaciónes del NLP ocurrió en 1954 e involucro la traducción de 60 oraciones rusas al inglés.
En los 1960´s ELIZA y SHRDLU fueron sistemas capaces de comunicarce con humanos.
Existen al menos 4 algoritmos de parsing:

1. El algoritmo CYK (Cocke–Younger–Kasami)

1.1. Solo trabaja con CFG en la forma normal de chumsky

1.2. El peor de los casos es $O(n^3.|G|)$ time. Donde $n$ es el tamaño de la cadena y $|G|$ es el tamaño de la gramática $G$ lo que lo hace uno de los mas eficientes en el peor de los casos.

2. Earley parser

2.1. Trabaja con todas las gramáticas libres del contexto.

2.2. Tiene un tiempo de ejecución $O(n^3)$ en el caso promedio,$O(n^2)$ en gramáticas no ambiguas y $O(n)$ para CFG deterministas.

3. LR parser (Left-to-right, Rightmost derivation in reverse)
Son tipos de bottom-up parsers que analizan DCFG en tiempo lineal.

3.1. LALR parser

3.2. Canonical LR parser

3.3. Minimal LR parser

3.4. GLR parser

4. LL parser (Left-to-right, Leftmost derivation)
Es un top-down parser para un subset de CFG

# 5. Propuestas

- Implementar y analizar dos algoritmos de parseo para construir una gramática independiente del contexto bajo una gramática y un lenguaje dado.
- Analizar posibles implementaciones y soluciones en casos de ambigüedad y presentar propuestas utilizadas actualmente por grandes empresas como Amazon o Google.

# 6. Implementación de Algoritmos

Para implementar los algoritmos CYK y Earley definimos primero que es una gramática.

## 6.1. Clase Gramática

Para la implementación de la estructura regla, la dividimos en 2 partes: Derecha e Izquierda. Ejemplo: izq ->der. La parte izquierda estará definida por un string y la derecha por un vector de strings. Asimismo, creamos la clase gramática que almacenará en un vector de punteros a reglas, todas las reglas a analizar.

```
#ifndef C___GRAMATICA_H
#define C___GRAMATICA_H

#include <string>
#include <utility>
#include <vector>
#include <iostream>


using namespace std;


struct regla{
    string izq;
    vector<string> der;
    regla(string izq, vector<string> der) : izq(std::move(izq)), der(
   std::move(der)) {}
    string preattyStringOutput();
};

class gramatica{
    vector<regla*> gram;

public:
    gramatica() = default;
    void crearRegla(string izq, const string& der);
  regla* inicio();
  regla* get_regla(char);

    string queReglaDeriva(const string& x);

    void print();

```

```
33      virtual ~gramatica();
34  };
35
36  #endif //C___GRAMATICA_H
```

## 6.2. Algoritmo CYK

El siguiente algoritmo, considerado 'bottom-up parsing', fue inventado por John Cocke, Daniel Younger y Tadao Kasami. Este solo opera con gramáticas en la forma normal de Chomsky (6).

Utilizando la notación Big O, el peor caso del algoritmo CYK es $\mathcal{O}\left(n^3 \cdot |G|\right)$ donde $n$ es la longitud de la cadena y $|G|$ es el tamaño de la gramática (10). Esta complejidad hace que sea uno de los algoritmos más eficientes de parseo en términos del peor caso.

Cumpliendo con los estándares, utilizaremos programación dinámica para resolver el Algoritmo CYK. Conforme al pseudocódigo expuesto en la Figura 1, realizaremos la implementación en C++, orientada a objetos.

```
CYK(G, S, w = a_1 a_2 ... a_n)
 1: if w = ϵ AND existe regla S → ϵ
 2:     return VERDADERO
 3: for i = 1 to n
 4:     for cada regla A → a_i
 5:         M[i, i] = M[i, i] ∪ {A}
 6: for ℓ = 2 to n
 7:     for i = 1 to n − ℓ + 1
 8:         j = i + ℓ − 1
 9:         for k = i to j − 1
10:             if existe regla A → BC, B ∈ M[i, k],
                C ∈ M[k + 1, j]
11:                 M[i, j] = M[i, j] ∪ {A}
12: return (S ∈ M[1, n])
```

Figura 1: Pseudocódigo del algoritmo CYK

Nosotros creamos un array llamado matriz para almacenar el resultado ya procesado. Asimismo, en el método 'solve', ejecutaremos el algoritmo. Dicho método retornará un doble puntero, al puntero del array donde estará guardado nuestro resultado dinámico con la solución. También, en el método 'cadenaAceptada()', retornamos TRUE si la cadena es aceptada por el algoritmo, de lo contrario, retornaremos FALSE.

```
1 #ifndef C___CYK_H
2 #define C___CYK_H
```

```
3
4  #include <string>
5  #include <utility>
6  #include <vector>
7  #include <iostream>
8  #include <iomanip>
9  #include <unordered_map>
10
11 #include "gramatica.h"
12
13 using namespace std;
14
15
16 class CYK{
17 private:
18     gramatica* G;
19     string cadena;
20     char S;
21
22     string **matriz;
23     size_t arraySize;
24
25     static string getString(char x){ string s(1, x); return s; }
26     static vector<string> distributiva(const string& a, const string& b
       );
27     string eliminarDuplicados(string x);
28
29 public:
30     CYK(gramatica *g,char S, const string& cadena);
31
32     string **solve();
33     bool cadenaAceptada();
34     friend ostream & operator<< (ostream &out, const CYK &cyk);
35
36
37     virtual ~CYK();
38
39 };
40
41 #endif //C___CYK_H
```

## 6.3. Algoritmo Earley Parser

A diferencia del algoritmo CYK el algoritmo de Earley es del tipo 'top-down' ya que éste evalúa si la cadena pertenece a la gramática empezando por la regla inicial y derivándola hasta obtener la cadena en el caso que ésta exista en la gramática.

Para los siguientes párrafos $\alpha$, $\beta$ y $\gamma$ representan cualquier cadena de variables y símbolos de la gramática. $X$ y $Y$ representan variables de la gramática.Y $a$ representa cualquier símbolo de la gramática.

Para lograr esta derivación se introduce una notación llamada 'dot-notation' o notación-punto $X \to \alpha \cdot \beta$. Esta notación indica que parte de una regla de la gramática ya ha sido reconocida (la parte a la izquiera del punto) y qué variable o símbolo se espera a continuación (la parte a la derecha del punto).

Dada una cadena de tamaño $n$ se construye un array de tamaño $n + 1$ donde cada posición en el array a partir de la posición 1 corresponde a un símbolo en la cadena correspondiendo la posición 0 al $\epsilon$ que precede a la cadena. Cada lugar en el array le corresponde a un conjunto de estados donde cada estado es una tupla que se construye a partir de una 'dot-notation' y la posición en el array donde ésta se originó $(X \to \alpha \cdot \beta, i)$. Llamamos al conjunto de estados que se encuentran en la posición $k$ $E(k)$.

Este algoritmo consiste en repetir tres operaciones basadas en estos estados: predecir, escanear y completar.

- Predecir: para todos los estados en $E(k)$ de la forma $(X \to \alpha \cdot Y\beta, i)$, agregamos $(Y \to \cdot\gamma, k)$ a $E(k)$ para todas las producciones de la forma $Y \to \gamma$ en la gramática.
- Escanear: si $a$ es el siguiente símbolo a leer entonces para todos los estados en $E(k)$ de la forma $(X \to \alpha \cdot a\beta, i)$, agregamos $(X \to \alpha a \cdot \beta, i)$ a $E(k + 1)$.
- Completar: para todos los estados en $E(k)$ de la forma $(Y \to \gamma\cdot, i)$, buscamos todos los estados en $E(i)$ de la forma $(X \to \alpha \cdot Y\beta, j)$ y agregamos $(X \to \alpha Y \cdot \beta, j)$ a $E(k)$.

Notese que el tamaño de $E(k)$ puede aumentar de tamaño después de cada operación así que se tiene que realizar las respectivas operaciones a los nuevos estados generados por operaciones previas.

```
#ifndef __Ealey_Parcer__
#define __Ealey_Parcer__

#include "gramatica.h"

struct Estado{
  size_t punPos;//marca que parte de la regla ya a sido parceada
  size_t origen;//en que etapa del algoritmo fue creado

  string izq;
  string der;

  char siguiente(){return der[punPos];}
  bool esta_completo(){return punPos==der.size();}

  void mostrar();

  Estado(regla* _regla, size_t cual,size_t ppos,size_t ori);

  Estado(Estado* otro_estado,size_t pos);

  bool es_igual_a(Estado* otro_estado);
};

class Earley{
  vector<vector<Estado*> > tabla;
  gramatica* _gramatica_;
  string _cadena_;

  void agregar_estado(regla*,size_t,size_t,size_t);//a partir de una
   regla
  void agregar_estado(Estado*,size_t,size_t);//a partir de otro estado;
   avanza el punto

  void predecir(Estado*,size_t);
  void escanear(Estado*,size_t);
  void completar(Estado*,size_t);

public:
  Earley(gramatica*,string);
  bool reconocer();
    virtual ~Earley();
};

bool es_mayuscula(char letra){return (letra>='A' && letra<='Z');}

```

```
45 #endif
```

# 7. GitHub

Nuestro proyecto se encuentra almacenado en el repositorio GitHub. Para acceder, podrá hacer **click aquí** .

# Referencias

[1] B. Box, *Natural Language Processing 5 3 Context Free Grammars Part 1 1211*, 2018. Disponible en https://www.youtube.com/watch?v=VkXSpWc_8FM.

[2] A. McCallum, *Context Free Grammars, Introduction to Natural Language Processing*, 2017. Disponible en https://people.cs.umass.edu/~mccallum/courses/inlp2007/lect5-cfg.pdf.

[3] B. College, *Overview of NLP: Issues and Strategies*. Disponible en http://www.bowdoin.edu/~allen/nlp/nlp1.html.

[4] E. Shutova, *Context-free grammars and parsing*, 2015. Disponible en https://www.cl.cam.ac.uk/teaching/1516/L90/slides4-public.pdf.

[5] Wikipedia, *Context-free grammar*. https://en.wikipedia.org/wiki/Context-free_grammar.

[6] Wikipedia, *CYK Algorithm*. https://en.wikipedia.org/wiki/CYK_algorithm.

[7] Wikipedia, *Earley parser*. https://en.wikipedia.org/wiki/Earley_parser.

[8] Wikipedia, *LR parser*. https://en.wikipedia.org/wiki/LR_parser.

[9] Wikipedia, *LL parser*. https://en.wikipedia.org/wiki/LL_parser.

[10] U. J. D. Hopcroft, John E., *Introduction to Automata Theory, Languages, and Computation*. https://archive.org/details/introductiontoau00hopc.