

---

**TD2 - Friday, October 2**


---

# 1 Syntax Analysis

Note : Good resources for this TD can be found at <http://www.labri.fr/perso/soueidan/compilation/>

## 1.1 Cocke-Younger-Kasami (CYK) Algorithm

The Cocke-Younger-Kasami algorithm determines whether a string can be generated by a given context-free grammar and, if so, how it can be generated. The algorithm is an example of dynamic programming. The standard version of CYK can only recognize languages defined by context-free grammars in Chomsky Normal Form (CNF). However, since any context-free grammar can be converted to CNF without too much difficulty, CYK can be used to recognize any context-free language.

For the sake of simplicity, in what follows we consider grammars where each production has at most two symbols. Example - grammar  $G$  :

$$S \rightarrow SS$$

$$S \rightarrow Ab$$

$$A \rightarrow aS$$

$$S \rightarrow ab$$

Let  $w$  be a string. We define the boolean array  $P_w[\alpha, i, j]$  where  $\alpha$  is a grammar symbol and  $1 \leq i \leq j \leq |w|$  in the following way :

The boolean value  $P_w[\alpha, i, j]$  means that  $\alpha$  derives the substring of  $w$  consisting of the  $i$ th through the  $j$ th symbols in zero or more steps. Hence we have :

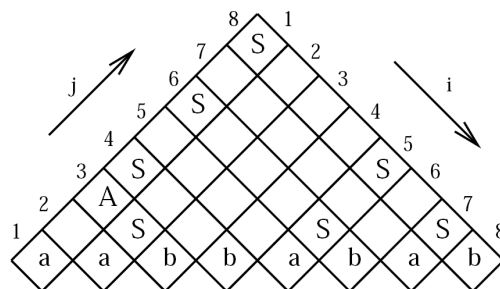
-  $\alpha$  is a terminal : if  $\alpha$  is the  $i$ th symbol of  $w$  then  $P_w[\alpha, i, i] = true$ , else  $P_w[\alpha, i, i] = false$ . Also,  $P_w[\alpha, i, j] = false$  for all  $i < j$ .

-  $\alpha$  is a nonterminal :  $P_w[\alpha, i, j] = true$  if there exists some production  $\alpha \rightarrow \beta\gamma$  and some  $i \leq k < j$  s.t.  $P_w[\beta, i, k] = true$  and  $P_w[\gamma, k + 1, j] = true$ , or if there exists a production  $\alpha \rightarrow \gamma$  s.t.  $P_w[\gamma, i, j] = true$ .

How many steps are necessary to compute  $P_w[\alpha, i, j]$  for one  $\alpha$ ?

What is the complexity to compute  $G$ ?

What is the condition that  $G$  generates  $w$ ? Example :  $w = aabbabab$  and the grammar given above. The CYK array is filled in.



Then, consider the context-free grammar  $G$  :

$$\begin{aligned} S &\rightarrow a \\ S &\rightarrow iS \\ S &\rightarrow iSeS \end{aligned}$$

Replace  $G$  with an equivalent context-free grammar  $G'$  that has no more than two symbols on the right side of the production. Now, apply the CYK algorithm for the string  $w = iaeiiaea$  (you can use a figure similar to the example).

## 1.2 Top Down Analysis : LL(1)

**Exercise 1** Eliminate left recursions and left factors for the following grammar.

$$\begin{aligned} S &\rightarrow AEB \\ A &\rightarrow Ax|Ay|Ba|a \\ E &\rightarrow = | \neq \\ B &\rightarrow Ab|b \end{aligned}$$

**Exercise 2** Consider the following grammar. Note that `id`, `+`, `[`, `]`, and `“,”` are terminals.

$$\begin{aligned} E &\rightarrow E + T | T \\ T &\rightarrow id | id[] | id[X] \\ X &\rightarrow E, E | E \end{aligned}$$

- Eliminate left recursion in the grammar.
- Perform left factoring for the grammar.
- Compute the **First** set for all symbols in the grammar.
- Compute the **Follow** set for all non-terminals in the grammar.
- Build an LL(1) parser for the grammar.
- Parse the string `id + id[id + id, id[]]`. Show the stack, the input, and the action taken.
- Build the parse tree while you are parsing. Show your parse tree.

**Exercise 3** Consider the following grammar for postfix expressions :

$$\begin{aligned} E &\rightarrow EE+ \\ E &\rightarrow EE* \\ E &\rightarrow \text{num} \end{aligned}$$

- a. Eliminate left-recursion in the grammar.
- b. Do left-factorisation of the grammar produced in question a.
- c. Calculate **Nullable** (the nonterminals that can generate the empty string  $\epsilon$ ), **FIRST** for every production and **FOLLOW** for every nonterminal in the grammar produced in question b.
- d. Make a LL(1) parse-table for the grammar produced in question b.

**Exercise 4** Consider the following grammar :

$$S \rightarrow uBDz$$

$$B \rightarrow Bv|w$$

$$D \rightarrow EF$$

$$E \rightarrow y|\epsilon$$

$$F \rightarrow x|\epsilon$$

Calculate Nullable, FIRST and FOLLOW sets.

Construct the LL(1) parse-table and give evidence that this grammar is not LL(1).

Give an LL(1) grammar that accepts the same language and built the LL(1) parse table for it.

### 1.3 Bottom Up Analysis : LR(0)

**Exercise 5**

Given the grammar

$$S \leftarrow AB\$$$

$$A \leftarrow aA$$

$$A \leftarrow x$$

$$B \leftarrow bB$$

$$B \leftarrow c$$

- Construct the automaton that can represent this grammar (also called Characteristic Finite State Machine for the grammar). Is it LR(0)? Why or why not?
- Construct action and goto tables from the automaton.
- Show the actions of a parser (using the action and goto tables) when parsing the string "aaxbc\$".

### 1.4 TP : YACC

**Exercise 6** You were hired to write an interpreter for a simple programming language called CATMOUSE for simulating cat and mouse games. This language allows the programmer to specify the starting positions and movements of cats and mice. The CATMOUSE programming language has a program definition (shown first) and seven types of statements :

Statement	Meaning
size i j begin <i>stmts</i> halt	program definition - defines height i and width j of the room.
cat v i j d;	draw a cat at position (i, j) in direction d
mouse v i j d;	draw a mouse at position (i, j) in direction d
hole i j;	draw a hole at position (i, j)
move v;	move the critter one space in its current direction
move v i;	move the critter i spaces in its current direction
clockwise v;	turn the critter v 90 degrees clockwise
repeat i <i>stmts</i> end;	execute <i>stmts</i> i times

where v is a variable, i and j are integers, d is a direction (north, south, east or west) and *stmts* represents 1 or more valid statements.

Here is a sample CATMOUSE program that draws a cat, mouse, and hole and then moves the critters around the room. We will assume that all rooms are a grid of points (x,y) with width w and height h with x from 0 to width and y positions from 0 to height. The upper left point is point (0,0). A comment is anything from // to the end of the line. The language is not case sensitive.

```
size 40 60
begin // room has height 40 and width 60
  cat charlotte 12 1 west; // a cat, charlotte, at (12,1) heading west
  mouse 1A 3 10 east; // a mouse, 1A, (3,10) heading east
  hole 9 10; // a hole at (9,10)
  repeat 3
    move charlotte; // charlotte moves one step west
    move 1A; // 1A moves one step east
  end;
  clockwise charlotte;
  clockwise charlotte;
  clockwise charlotte; // charlotte now heading south
  repeat 3
    move 1A; // 1A moves one step east
    move charlotte 3; // charlotte moves three steps west
  end;
halt
```

In this exercise, build the CATMOUSE language using the LEX and YACC tools. Use LEX for the lexical analysis - identify the elementary parts (tokens) of a CATMOUSE program. Then, use YACC to identify syntactically correct CATMOUSE programs.

1. Lexical analysis part
  - What are the keywords of a CATMOUSE program?
  - What about variables? Note : Valid variable names may contain at least one letter and any number of digits. Keywords are not variables. So, *east* is a keyword, not a variable!
  - Valid integers may contain up to three digits (0-9). If it starts with 0 then it must be of length 1.
  - There is only one punctuation symbol ";" in the CATMOUSE language.
2. Syntactic analysis part
  - What is the grammar for the CATMOUSE Language?
  - Use YACC to identify syntactically correct CATMOUSE programs.