Grammaire simplifiée

expr ::= concat "|" expr

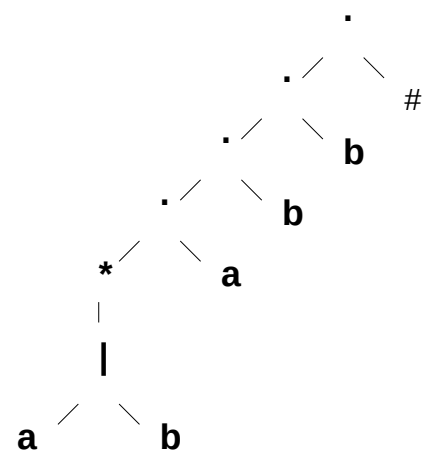
concat ::= repet concat

repet ::= simple "*" |
simple

simple ::= "(" expr ")" |
car

car ::= tout sauf "|", "*", "(", ")"|
"\|" | "*" | "\(" | "\)"

Arbre pour **(a|b)*abb**



#: caractère de fin

Analyse récursive descendante
Parsing

```
typedef enum { PIPE, STAR, LBRACE,
               RBRACE, CAR, END} TOKEN;

TOKEN token;
char token_value;

next_token() {
 char c = getchar();
 token_value=c;
 if ((c==EOF) || (c=='\n')) token=END;
 else {
  switch (c) {
    case '|': token = PIPE; break;
    case '*': token = STAR; break;
    ...
    }
  }
  }
```

Analyse récursive descendante

```
#include "parse.h"
typedef enum {ALTER, CONCAT, REPET, LETTER} TYPENODE;
typedef struct node {
  TYPENODE type;
  char value;
  struct node *left, *right;} NODE;

NODE *root;

NODE *expr() {
  NODE *child, *node;
  child=concat(); if (token == END) return child;
  if (child) == NULL) return_error();
  if (token==PIPE) {
    créer node type ALTER;
    next_token();
    if ((node->right=expr())==NULL) return_error();
    return node;
  }
  else return child;
}

NODE *concat() {
  NODE *child, node;
  child=repet(); if (token==END) return child;
  if (child==NULL) return_error();
  if ((token==LBRACE) || (token==CAR)) {
    créer node type CONCAT;
    if ((node->right=concat())==NULL) return_error();;
    return node;
  }
  else return child;
}

NODE *repet() {
  NODE *child, node;
  child=simple(); if (token==END) return child;
  if (child==NULL) return_error();
  if (token==STAR) {
    créer node type STAR;
    next_token();
    return node;}
  else
    return child;
}

NODE *simple() {
  NODE *child, node;
  if (token == LBRACE) {
    next_token();
    if ((child=expr())==NULL) return_error();
    if (token!=RBRACE) return_error();
    next_token();
    return child;
  } else {
    if (token==END) return NULL;
    if (token!=CAR) return_error();
    créer node type LETTER;
    next_token();
    return node;
  }
}
```