Pratique du C (INFO 301) : Projet : implantation du filtre grep

3 novembre 2011

1 Introduction

On se propose d'implanter en C-ANSI un filtre `mongrep` inspiré du filtre `grep` existant sur la plupart des systèmes Unix.

Bien que certaines notions utilisées dans ce projet vous sont familières puisque vues en cours de compilation, le sujet est autosuffisant et ne nécessite pas la maîtrise de ce cours.

Le filtre `mongrep` imprime les lignes, appartenant à un fichier, qui contiennent un certain *motif*. Ce dernier correspond à l'ensemble des mots d'un *langage* reconnu par un automate associé à une *expression rationnelle*.

Le motif recherché est donné par l'utilisateur dans la ligne de commande sous la forme d'un paramètre représentant une expression rationnelle (cf. section 5 page 9).

Le filtre `mongrep` doit donc reconnaître le langage associé pour effectuer la recherche du motif.

Pour toute expression rationnelle, il existe un unique (au nom des états près) automate fini déterministe minimal *reconnaissant* le langage associé. La construction d'un tel automate passe par

1. la construction d'un arbre de syntaxe abstraite associé à l'expression rationnelle ;
2. la construction d'un automate fini à partir de cet arbre ;

Nous allons coder ces phases qui nous seront suffisantes (sans aller jusqu'à construire un automate fini déterministe).

2 Une grammaire d'expressions rationnelles

La grammaire des expressions rationnelles utilisées par la commande `mongrep` est donnée par la Figure 1 page 2. C'est une version très simplifiée de celle utilisée par `grep`. Notre filtre `mongrep` va prendre une chaîne de caractères codant une expression rationnelle en paramètre depuis la ligne de commande d'un shell. Dans un premier temps, cette chaîne va être convertie en un arbre binaire dénommé *arbre de syntaxe abstraite*.

3 Construction d'un arbre de syntaxe abstraite correspondant à une expression rationnelle

La Figure 2 page 3 schématise la représentation d'un tel arbre et les principales fonctions permettant sa construction à partir d'une chaîne de caractères — prise sur l'entrée standard — pour une grammaire simplifiée. Pour être plus précis, la fonction `simple` de la Figure 2 page 3 implante la règle

```
simple ::= '(' expr ')' || car
```

de la grammaire simplifiée.

Notez bien qu'en ce qui concerne le filtre `mongrep` que l'on vous demande d'implanter, la chaîne de caractères représentant l'expression rationnelle considérée n'est pas prise sur l'entrée standard — comme

```
expression ::= expression_concatenation '|' expression || expression_concatenation
expression_concatenation ::= expression_repetition expression_concatenation
                           || expression_repetition
expression_repetition ::= expression_simple '*' || expression_simple
expression_simple ::= '(' expression ')' || car_non_speciaux || intervalle
car_non_speciaux ::= tout caractere sauf '|', '*', '[', ']', '.' || '\|' || '\*'
                   || '\[' || '\]' || '\.'
intervalle ::= '.' || '[' liste ']' || '[^' liste ']' || '[' liste '-]'
             || '[^' liste '-]'
liste ::= non_moins liste1
liste1 ::= non_fermant liste1
non_moins ::= tout caractere sauf '-'
non_fermant ::= tout caractere sauf ']'
```

FIGURE 1 – Grammaire des expressions rationnelles de `mongrep`

dans l'exemple de la Figure 2 — mais comme paramètre du filtre. Ainsi, on vous demande d'adapter le code de la Figure 2 page 3 afin d'être utilisable dans le filtre `mongrep` et de le compléter afin de pouvoir implanter la grammaire de la Figure 1 page 2.

Les algorithmes présentés dans la suite utilisent l'arbre de syntaxe abstraite construit précédemment, auquel on a ajouté un nœud racine (de type `CONCAT`) ayant pour fils droit un nœud de fin end (noté # dans la Figure 2) et pour fils gauche l'arbre de syntaxe abstrait originel.

4 Construction d'un automate non déterministe à partir d'un arbre de syntaxe abstraite

Illustrons la méthode à l'aide d'un exemple. Considérons l'expression rationnelle :

$$(ab + c)^\star ab$$

correspondant à l'arbre de syntaxe de la Figure 3 page 4. Le but de l'algorithme est de construire l'automate de la Figure 4 page 4. Commençons par introduire un peu de terminologie.

Définition 1 *Etant donnée une expression rationnelle, on définit une* position *dans cette expression comme l'indice d'un des caractères alphabétiques la composant. Pour l'expression* $(ab + c)^\star ab$*, il y a cinq positions :*

$$1)\, a, \quad 2)\, b, \quad 3)\, c, \quad 4)\, a, \quad 5)\, b.$$

L'automate de la Figure 4 page 4 a les caractéristiques suivantes :
- un état initial O ;
- un état par positions (i.e. caractère alphabétique de l'expression rationnelle) ;
- depuis l'état initial, on peut se rendre sur les états correspondant aux positions où peut commencer un mot vérifiant l'expression rationnelle ($1, 3$ ou 4) ;
- les états terminaux correspondent aux positions où cette expression peut se terminer. En l'occurence, l'expression considérée ne peut se terminer qu'après le 'b' final (position 5).

Interprétation de l'automate. Informellement, l'automate permet de coder l'information suivante :

étant dans l'état i, quelle peut être la lettre suivante, et en quelle position m'amènera-t-elle ?

Par exemple, depuis l'état 2 (dans la lecture de l'expression rationnelle, on vient de lire le premier b) :
- on peut lire un c (application de l'étoile, choix de c) et arriver en position 3 ;

Grammaire simplifiée

expr ::= concat "|" expr

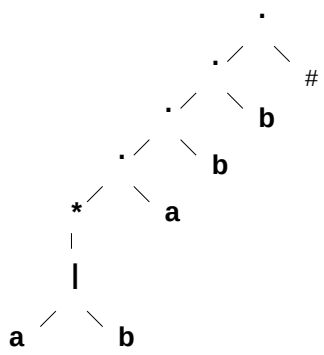
concat ::= repet concat

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;
  }
}
```

FIGURE 2 – Construction de l'arbre

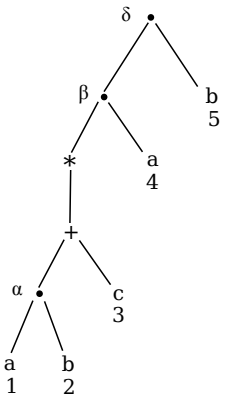


FIGURE 3 – Arbre syntaxique de l'expression rationnelle $(ab + c)^\star ab$

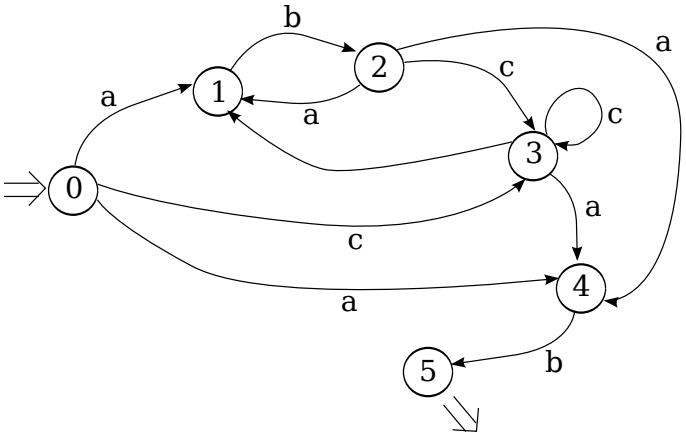


FIGURE 4 – automate non déterministe construit à partir de l'expression rationnelle $(ab + c)^\star ab$

– on peut lire un a (application de l'étoile, choix de ab) et revenir en position 1 ;
– on peut lire un a (sortie de l'étoile, lecture du caractère suivant) et arriver en position 4.

Chacune de ces options correspond à une transition dans l'automate, qui représente ainsi toutes les transitions possibles engendrées par ce raisonnement.

Indicateurs. Avant d'automatiser le processus, et trouver un algorithme permettant de construire l'automate à partir de l'expression rationnelle, on peut dégager trois notions importantes :
– par quels caractères peut commencer un mot vérifiant une expression rationnelle donnée ? (ou dit autrement, par quelle position peut-on 'entrer' dans une expression rationnelle ?)
– par quels caractères (quelles positions) peut se terminer un mot vérifiant une expression rationnelle ? (par quelle position peut-on 'sortir' d'une expression rationnelle ?)
– à partir d'une position donnée, quelles sont les positions accessibles ?

Il existe une dernière notion à envisager, qui n'est pas illustrée par notre exemple :
– à partir d'une position, peut-on reconnaître le mot vide ϵ ?

Définition 2 *Ces quatre notions (indicateurs) seront nommées dans l'ordre :*
– *first*(i) *: ensemble des positions par lesquelles peut commencer une expression rationnelle ;*
– *last*(i) *: ensemble des positions où peut se terminer une expression rationnelle ;*
– *follow*(i) *: ensemble des positions accessibles depuis la position* i *;*
– *nullable*(i) *: vrai si* ϵ *est reconnu depuis cette position.*

L'arbre de syntaxe découpe l'expression rationnelle en sous-expressions. Les quatre indicateurs peuvent donc être définis en chaque nœud de l'arbre, puisque chacun de ces nœuds est lui-même la racine d'une expression rationnelle.

4.1 Indicateurs pour les feuilles

Les feuilles sont de trois types $\emptyset, \epsilon$ ou i ; les valeurs des indicateurs pour chaque type de feuille se définissent facilement :
– Nœud $\emptyset$: nullable = false, first = last = follow = $\emptyset$;
– Nœud ϵ : nullable = true, first = last = follow = $\emptyset$;
– Nœud position i : nullable = false, first(i) = last$(i) = \{i\}$, follow$(i) = \emptyset$.

4.2 Indicateurs pour les nœuds internes

Il y a trois types de nœuds internes : concaténation, étoile, union. Pour chacun de ces types de nœud, on peut calculer les indicateurs correspondants, à partir des indicateurs de leur(s) descendant(s).

4.2.1 Nœud concaténation

Dans cette section, nous allons considérer comment déterminer les indicateurs d'un nœud concaténation.

nullable. Un nœud concaténation peut engendrer ϵ si son fils gauche (g) et son fils droit (d) sont capables d'engendrer ϵ :

$$\text{nullable}(g \bullet d) = \text{nullable}(g) \text{ et } \text{nullable}(d).$$

first. Pour un nœud concaténation, les positions pouvant commencer un mot sont :
– celles pouvant commencer son fils gauche ;
– et seulement si son fils gauche peut engendrer ϵ, les positions pouvant débuter un mot du fils droit.

$$\text{first}(g \bullet d) = \begin{cases} \text{first}(g) \text{ si nullable}(g) = \text{false}, \\ \text{first}(g) \cup \text{first}(d) \text{ si nullable}(g) = \text{true}. \end{cases}$$

last. Les positions susceptibles de terminer un mot d'un nœud concaténation sont :
- celles qui peuvent terminer son fils droit ;

$$\text{last}(g \bullet d) = \text{last}(d) \text{ si } \text{nullable}(d) = \text{false},$$

- auxquelles on ajoute celles terminant son fils gauche, si le fils droit peut engendrer ϵ :

$$\text{last}(g \bullet d) = \text{last}(g) \cup \text{last}(d) \text{ si } \text{nullable}(d) = \text{true}.$$

Donc, on a :

$$\text{last}(g \bullet d) = \begin{cases} \text{last}(d) \text{ si } \text{nullable}(d) = \text{false}, \\ \text{last}(g) \cup \text{last}(d) \text{ si } \text{nullable}(d) = \text{true}. \end{cases}$$

follow. Soit x une position dans $g \bullet d$.
- Si x est dans d, les positions qui peuvent lui succéder sont les mêmes que celles qui peuvent lui succéder dans d :

$$\text{follow}(g \bullet d, x) = \text{follow}(d, x) \ ;$$

- Si x est dans g, mais n'appartient pas à $\text{last}(g)$, les positions qui peuvent lui succéder sont les mêmes que celles qui peuvent lui succéder dans g :

$$\text{follow}(g \bullet d, x) = \text{follow}(g, x) \ ;$$

- Si x est dans $\text{last}(g)$, les positions suivant x sont celles lui succédant dans g, auxquelles on ajoute toutes les positions pouvant commencer d :

$$\text{follow}(g \bullet d, x) = \text{follow}(g, x) \cup \text{first}(d).$$

4.2.2 Nœud union

nullable. Un nœud union $(g + d)$ permet d'engendrer ϵ si l'un de ses fils au moins peut le faire :

$$\text{nullable}(g + d) = \text{nullable}(g) \text{ or } \text{nullable}(d).$$

first. Les positions pouvant commencer un mot défini par un nœud union sont celles permettant de commencer un mot de g plus celles pouvant comencer un mot de d :

$$\text{first}(g + d) = \text{first}(g) \cup \text{first}(d).$$

last. Même raisonnement pour les fins de mots :

$$\text{last}(g + d) = \text{last}(g) \cup \text{last}(d).$$

follow. Si x est une position de $g + d$, elle est soit une position de g, soit une position de d. On a donc simplement :

$$\text{follow}(g + d, x) = \begin{cases} \text{follow}(g, x) & \text{si } x \in g, \\ \text{follow}(d, x) & \text{si } x \in d. \end{cases}$$

4.2.3 Nœud étoile

nullable. Une expression à l'étoile peut engendrer epsilon :

$$\text{nullable}(\star) = \text{true}.$$

first et last. Les positions pouvant commencer un mot correspondant à une expression à l'étoile sont exactement celles qui permettent de commencer un mot du langage de base. Idem pour les positions en fin d'expression :

$$\text{first}(g^\star) = \text{first}(g), \qquad \text{last}(g^\star) = \text{last}(g).$$

follow. Si $x \notin \text{last}(g)$, alors les positions qui peuvent lui succéder sont celles qui pouvaient lui succéder dans g. Sinon, il faut y ajouter les débuts possibles de mots de g, qui viendront se concaténer derrière x :

$$\text{follow}(g^\star, x) = \begin{cases} \text{follow}(g, x) & \text{si } x \notin \text{last}(g), \\ \text{follow}(g, x) \cup \text{first}(g) & \text{si } x \in \text{last}(g). \end{cases}$$

4.3 Exemple d'application

On reprend l'arbre de syntaxe de la Figure 3 page 4, où figurent les numéros des positions. Nous partons des feuilles pour remonter à la racine. Lorsque vous le programmerez, une approche récursive sera plus facile à concevoir.

Pour la position 1 :

$$\begin{array}{lcl} \text{nullable}(1) & = & \text{false}, \\ \text{first}(1) & = & \{1\}, \\ \text{last}(1) & = & \{1\}, \\ \text{follow}(1) & = & \emptyset. \end{array}$$

Pour la position 2 :

$$\begin{array}{lcl} \text{nullable}(2) & = & \text{false}, \\ \text{first}(2) & = & \{2\}, \\ \text{last}(2) & = & \{2\}, \\ \text{follow}(2) & = & \emptyset. \end{array}$$

Pour le nœud interne $1 \bullet \alpha 2$:

$$\begin{array}{lcl} \text{nullable} & = & \text{false}, \\ \text{first} & = & \{1\}, \\ \text{last} & = & \{2\}, \\ \text{follow}(1) & = & \{2\}, \\ \text{follow}(2) & = & \emptyset. \end{array}$$

Pour la position 3 :

$$\begin{array}{lcl} \text{nullable}(3) & = & \text{false}, \\ \text{first}(3) & = & \{3\}, \\ \text{last}(3) & = & \{3\}, \\ \text{follow}(3) & = & \emptyset. \end{array}$$

Pour le nœud $+$:

$$\begin{array}{lcl} \text{nullable} & = & \text{false}, \\ \text{first} & = & \{1, 3\}, \\ \text{last} & = & \{2, 3\}, \\ \text{follow}(1) & = & \{2\}, \\ \text{follow}(2) & = & \emptyset, \\ \text{follow}(3) & = & \emptyset. \end{array}$$

Pour le nœud interne $1 \star 2$:

$$\begin{array}{lcl} \text{nullable} & = & \text{true}, \\ \text{first} & = & \{1,3\}, \\ \text{last} & = & \{2,3\}, \\ \text{follow}(1) & = & \{2\}, \\ \text{follow}(2) & = & \{1,3\}, \\ \text{follow}(3) & = & \{1,3\}. \end{array}$$

Pour la position 4 :

$$\begin{array}{lcl} \text{nullable}(4) & = & \text{false}, \\ \text{first}(4) & = & \{4\}, \\ \text{last}(4) & = & \{4\}, \\ \text{follow}(4) & = & \emptyset. \end{array}$$

Pour le nœud interne $1 \bullet \beta 2$:

$$\begin{array}{lcl} \text{nullable} & = & \text{false}, \\ \text{first} & = & \{1,3,4\}, \\ \text{last} & = & \{4\}, \\ \text{follow}(1) & = & \{2\}, \\ \text{follow}(2) & = & \{1,3,4\}, \\ \text{follow}(3) & = & \{1,3,4\}, \\ \text{follow}(4) & = & \emptyset. \end{array}$$

Pour la position 5 :

$$\begin{array}{lcl} \text{nullable}(5) & = & \text{false}, \\ \text{first}(5) & = & \{5\}, \\ \text{last}(5) & = & \{5\}, \\ \text{follow}(5) & = & \emptyset. \end{array}$$

enfin, au sommet de l'arbre, pour la position $\bullet\delta$

$$\begin{array}{lcl} \text{nullable} & = & \text{false}, \\ \text{first} & = & \{1,3,4\}, \\ \text{last} & = & \{5\}, \\ \text{follow}(1) & = & \{2\}, \\ \text{follow}(2) & = & \{1,3,4\}, \\ \text{follow}(3) & = & \{1,3,4\}, \\ \text{follow}(4) & = & \{5\}, \\ \text{follow}(5) & = & \emptyset. \end{array}$$

Ces dernières informations permettent de définir entièrement l'automate :
- l'état initial n'est pas final (le nœud au sommet de l'arbre n'est pas nullable) ;
- follow fournit la table de transition de l'automate, aidé par first qui décrit les transitions depuis l'état initial ;
- last donne la liste, ici réduite à un seul état, des états terminaux.

Il vous reste encore à définir une structure de données pour l'automate, ainsi que les fonctions permettant de s'y déplacer.

Attention. L'automate est non déterministe ; vous devrez gérer une liste d'états accessibles : un mot sera reconnu si un état final appartient à cette liste lorsque le mot aura été complètement lu.

5 Implantation de la commande `mongrep`

La commande `mongrep` a la syntaxe suivante :

```
mongrep  expression-rationnelle < fichier
```

en utilisant la grammaire des expressions rationnelles présentée dans la section 2. Cette commande effectue la recherche des motifs dans le fichier envoyé sur son entrée standard ; elle affiche les lignes qui contiennent un mot du langage défini par l'expression rationnelle.

Remarque : les caractères ASCII constituant les métacaractères sont d'abord interprété par le shell. Ainsi le filtre `mongrep` doit en tenir compte comme dans les exemples suivants — et équivalents — d'appels :

```
% ./mongrep '(a|b)*abb' < foo # l'accent aig\"u ' bloque l\'evaluation
% ./mongrep \(a\|b\)\*abb < foo
```

Le caractère ASCII `\` permet de bloquer l'évaluation du métacaractère par le shell et permet ainsi à la commande `mongrep` de le traiter. Votre filtre `mongrep` doit permettre de considérer un métacaractère comme un caractère ASCII normal sur le même principe (utilisation de `\`).

6 Réalisation

Vous devez rendre (par l'interface PROF) un répertoire contenant :
- les codes sources de votre projet ;
- un Makefile permettant la compilation de votre projet ;
- un fichier de test `testfile` et un script shell `test.sh` utilisant `mongrep` sur ce fichier test.

Vous pouvez adjoindre une description de votre implantation.