Licence d'informatique
Module de C/Unix

Projet C

`igrep` — Recherche indexée

Jean-Luc LEVAIRE

Décembre 1997

La commande `igrep` (*indexed grep*) proposée ici est inspirée de la commande `grep` existant sur la plupart des systèmes Unix. Sa principale particularité est qu'elle utilisera le fichier d'index créé par la commande `stfsindex` du projet Unix afin de minimiser son espace de recherche.

La commande `igrep` sera développée en C-ANSI, de façon *modulaire*.

Reconnaissance du langage associé à une expression rationnelle

La commande `grep` imprime les lignes, appartenant à un ensemble de fichiers, qui contiennent un certain motif. Le motif recherché est donné dans la ligne de commande sous la forme d'un argument représentant une expression rationnelle. La commande `igrep` doit elle aussi reconnaître le langage associé à une expression rationnelle pour effectuer la recherche du motif.

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

1. construction de l'arbre de syntaxe abstraite correspondant à l'expression rationnelle ;
2. construction d'un automate fini déterministe à partir de cet arbre ;
3. minimisation de l'automate déterministe ;

Construction de l'arbre correspondant à l'expression

La grammaire des expressions rationnelles utilisées par la commande `igrep` est donnée Fig. 1. C'est une version légérement simplifiée de celle utilisée par `grep`. La construction de l'arbre de syntaxe abstraite à partir d'une expression se fait par une analyse récursive descendante, dont le principe est illustré Fig 2 pour une grammaire simplifiée.

Construction de l'automate déterministe

Les algorithmes présentés dans la suite fonctionnent sur l'arbre de syntaxe abstraite construit précédemment, auquel on a ajouté un nœud racine ($root$) ayant pour fils droit un symbole de fin n'appartenant pas à l'alphabet de départ (voir Fig. 2), et pour fils gauche l'ast originel. Pour construire un automate déterministe à partir de cet arbre, il faut calculer pour chaque nœud n les trois ensembles $premier(n)$, $dernier(n)$, $suivant(n)$, et la fonction $nul(n)$ définis à la figure Fig. 3. Ces ensembles et cette fonction peuvent se calculer en réalisant un parcours de l'arbre en profondeur d'abord.

Pour obtenir l'automate déterministe à partir de ces ensembles, il reste à appliquer l'algorithme présenté Fig. 4. Cet algorithme construit un ensemble d'états $Detat$, chaque état représentant un ensemble de nœuds de l'arbre, et une table de transition $Dtran$, représentant les transitions de l'automate. Ainsi, si $Dtran[T, a] = U$, alors il y a une transition dans l'automate

```
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 ']'
```

FIG. 1 – Grammaire des expressions rationnelles de `igrep`

de l'état T à l'état U étiquettée par la lettre a. Les états de $Detat$ sont marqués pour indiquer qu'ils n'ont pas encore été examinés. L'état initial de l'automate est $premier(root)$, les états finaux sont tous les états contenant le nœud représentant le symbole de fin.

Minimisation de l'automate

La minimisation d'un automate déterministe M s'effectue en partitionnant l'ensemble de ses états. Le calcul de cette partition Π_{finale} se fait en appliquant itérativement l'algorithme de la figure 5 sur la partition courante Π. Chaque application de cet algorithme à Π calcule une nouvelle partition Π_{new}, et, lorsque $\Pi_{new} = \Pi$, on a trouvé la partition finale $\Pi_{finale} = \Pi_{new}$. Si $\Pi_{new} \neq \Pi$, on réitère l'algorithme de la figure 5 en prenant $\Pi = \Pi_{new}$. Initialement, la partition Π comporte deux groupes, le groupe des états acceptants (finaux), et les états non-acceptants.

Chaque groupe d'états de Π_{finale} représente en fait un état de l'automate minimal M'. Pour calculer les transitions de l'automate minimal, on choisit pour chaque groupe de Π_{finale} un état *représentant*. Les états de l'automate minimal seront les états représentants. Soit s un état représentant, et supposons qu'il y a une transition par la lettre a dans l'automate non minimal de s vers t. Soit r le représentant du groupe de t, alors on a une transition dans l'automate minimal de s vers r par la lettre a. L'état initial de M' sera le représentant du groupe contenant l'état initial de M, ses états finaux seront les représentants des groupes qui contiennent un état final de M. Il reste à supprimer les états morts (bouclant sur eux-mêmes pour toute lettre) non finaux, et les états non-atteignables depuis l'état initial.

Implémentation de la commande **`igrep`**

La commande `igrep` a la syntaxe suivante :

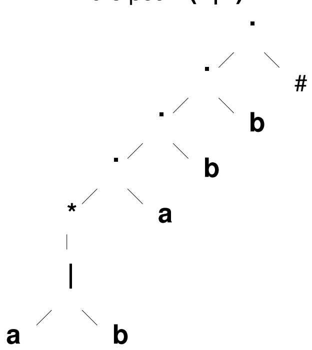
```
igrep [-z] expression-rationnelle [-f liste-de-fichiers]
```

La grammaire des expressions rationnelles est celle présentée précédemment. Sans l'option `-f`, la commande `igrep` effectue la recherche des motifs dans les fichiers indexés par la commande `stfsindex`. La liste de ces fichiers se trouve dans le fichier `$HOME/.stfs.filenames.igrep` utilise alors l'index pour optimiser sa recherche dans l'ensemble des fichiers indexés. Si l'option `-f` est présente, `igrep` se comporte comme la commande `grep`, c'est-à-dire qu'elle parcourt les fichiers passés en paramètre en affichant les lignes qui contiennent un mot du langage de l'expression rationnelle. Si il n'y a aucun fichier dans `liste-de-fichiers`, `igrep`

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

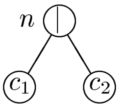
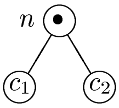
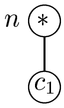
Analyse récursive descendante
Parsing

```
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;
  }
}
```

FIG. 2 – Construction de l'arbre

Nœud n	$premier(n)$	$dernier(n)$	$nul(n)$
n est une feuille étiquettée ϵ	$\emptyset$	$\emptyset$	**True**
n est une feuille étiquettée i	i	i	**False**
	$premier(c_1) \cup premier(c_2)$	$dernier(c_1) \cup dernier(c_2)$	$nul(c_1)$**ou** $nul(c_2)$
	si $nul(c1)$ alors $premier(c_1) \cup premier(c_2)$ sinon $premier(c_1)$	si $nul(c2)$ alors $dernier(c_1) \cup dernier(c_2)$ sinon $dernier(c_2)$	$nul(c_1)$**and** $nul(c_2)$
	$premier(c_1)$	$dernier(c_1)$	**True**

Construction de la fonction $suivant$: Soit n un nœud de l'arbre,

1. si n est un nœud de concaténation avec c_1 pour fils droit et c_2 pour fils gauche, et si i appartient à $dernier(c_1)$, alors tous les éléments de $premier(c_2)$ sont dans $suivant(i)$;
2. si n est un nœud de répétition, et si i appartient à $dernier(n)$, alors tous les éléments de $premier(n)$ sont dans $suivant(i)$;

FIG. 3 – Définition des fonctions $premier$, $dernier$, nul et $suivant$.

Initialement, $Detat$ contient un seul état non marqué constitué des nœuds de $premier(root)$.
Tant que il reste un état T non marqué dans $Detat$ **faire**
- marquer T ;
- **Pour** chaque lettre a de l'alphabet **faire**
  - Soit U l'ensemble des nœuds de l'arbre qui sont dans $suivant(p)$, avec p appartenant à T, et tel que p représente la lettre a ;
  - **Si** U n'est pas vide et n'appartient pas déjà à $Detat$, **alors** ajouter U à $Detat$ (sans le marquer) ;
  - $Dtran[T, a] = U$;

FIG. 4 – Construction de l'automate déterministe.

Pour chaque groupe G de Π **faire**
- Partitionner G en sous-groupes, de sorte que

  deux états s et t de G sont dans le même sous-groupe *si et seulement si*, pour toute lettre a de l'alphabet, les états s et t ont une transition par a vers des états qui appartiennent au même groupe de Π ;
- Remplacer G dans Π_{new} par l'ensemble des sous-groupes calculés précédemment ;

FIG. 5 – Algorithme de partitionnement d'une partition Π.

lit sur son entrée standard. Si l'option `-z` est présente, les fichiers compressés ou codés sont décompressés/décodés. Les utilitaires de décompression utilisés sont `uncompress` et `gunzip`. L'utilitaire de décodage considéré est `uudecode`.