Licence d'informatique
Module de Pratique du C

Travaux pratiques

Ma commande grep

Décembre 2007

Ce sujet est inspiré d'un projet élaboré par Jean-Luc Levaire.

Il s'agit d'implanter en C-ANSI un filtre `mgrep` inspiré du traditionnel filtre `grep`.

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.

1 Le filtre `mgrep`

Le filtre `mgrep` imprime les lignes de l'entrée standard qui contiennent un certain *motif*. Ce dernier correspond à l'ensemble des mots d'un *langage* associé à une *expression rationnelle*. Le motif recherché est donné dans la ligne de commande sous la forme d'un paramètre représentant une expression rationnelle. Le filtre `mgrep` 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 trois phases :

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

Nous allons coder ces phases bien que la dernière ne soit pas impérative à l'implantation du filtre `mgrep`.

La grammaire des expressions rationnelles utilisées par la commande `mgrep` est donnée par la figure 1. C'est une version très simplifiée de celle utilisée par `grep`.

Notre filtre `mgrep` va prendre une chaîne de caractères codant une expression rationnelle en paramètre depuis la ligne de commande. Dans un premier temps, cette chaîne va être convertie en un arbre binaire dénommé *arbre de syntaxe abstraite*.

```
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 `mgrep`

Id: tpgrep.tex,v 1.7 2007/12/19 10:58:25 marquet Exp
www.lifl.fr/~marquet/ens/pdc/tpgrep

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

La figure 2 schématise la représentation d'un arbre de syntaxe abstraite 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` implante la règle

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

de la grammaire simplifiée.

Notez bien qu'en ce qui concerne le filtre `mgrep` que l'on vous demande d'implanter, la chaîne de caractères représentant l'expression rationnelle considérée n'est pas prise sur l'entrée standard. Ainsi, on vous demande d'adapter le code de la figure 2 afin d'être utilisable dans le filtre `mgrep` et de le compléter afin de pouvoir implanter la grammaire de la figure 1.

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.

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

3.1 Construction d'ensembles d'états associés à chaque nœud

Pour construire un automate déterministe à partir de cet arbre, il faut calculer pour chaque nœud n les 5 ensembles alphabet(n), initial(n), final(n), suivant(n) et isnul(n) définis à la figure 3.

- isnul(n) est une valeur booléenne ;
- alphabet(n) est un ensemble de caractères ;
- les autres ensembles sont des ensembles d'états de l'automate.

Ces ensembles peuvent se calculer en réalisant un parcours de l'arbre en profondeur d'abord.

Remarques L'opération $\cup$ de réunion des ensembles vérifie la relation $\{i, j\} \cup \{i, k\} = \{i, j, k\}$. Les autres opérations sur les ensembles (intersection, complément...) ne sont pas utiles pour ce projet).

Vous êtes libres de choisir votre représentation des ensembles et des états (pour ces derniers, il peut être intéressant de leurs associer une étiquette unique pouvant être en relation avec l'adresse d'un nœud de l'arbre de syntaxe abstraite).

3.2 Construction de l'automate déterministe

Pour obtenir l'automate déterministe à partir de ces ensembles, il reste à appliquer l'algorithme présenté ci-dessous. 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 de l'état T à l'état U étiquetée par la lettre a. Les états de $Detat$ sont marqués pour pouvoir indiquer s'ils ont été examinés ou pas. L'état initial de l'automate est initial($root$), les états finaux sont tous les états associé au nœud représentant le symbole de fin.

Algorithme de construction de l'automate déterministe
Initialement, $Detat$ contient un seul état non marqué constitué des éléments de initial($root$).
Tant qu'il reste un état T non marqué dans $Detat$ **faire**

- marquer T;

Grammaire simplifiée

expr ::= concat "|" expr

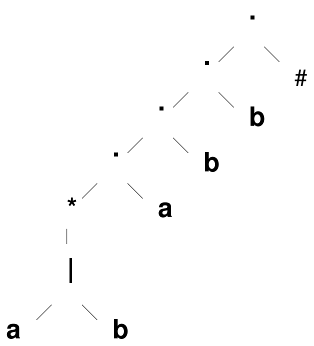
concat ::= repet concat

repet ::= simple "*" |
simple

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

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

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



#: caractère de fin

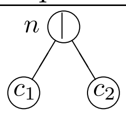
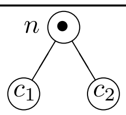
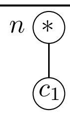
Analyse récursive descendante
Parsing

```
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	initial(n)	final(n)	isnul(n)
n est une feuille étiquetée end($= \#$)	$\{$end$\}$	$\{$end$\}$	**True**
n est une feuille étiquetée i	$\{i\}$	$\{i\}$	**False**
	initial(c_1) $\cup$ initial(c_2)	final(c_1) $\cup$ final(c_2)	isnul(c_1) **ou** isnul(c_2)
	si isnul(c_1) alors initial(c_1) $\cup$ initial(c_2) sinon initial(c_1)	si isnul(c_2) alors final(c_1) $\cup$ final(c_2) sinon final(c_2)	isnul(c_1) **and** isnul(c_2)
	initial(c_1)	final(c_1)	**True**

Soit n un nœud de l'arbre,

– Construction de l'ensemble suivant :

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

– Construction de l'ensemble alphabet :

1. si n est un nœud de concaténation ayant c_1 pour fils droit et c_2 pour fils gauche, alors alphabet(n) est la réunion de alphabet(c_1) et alphabet(c_2) ;
2. si n est un nœud de répétition, alors alphabet(n) est égal à alphabet(c_1).

FIG. 3 – Définition des ensembles alphabet, initial, final, isnul et suivant.

– Soit U l'ensemble des nœuds de l'arbre qui sont dans suivant(p), avec p le nœud associé à T.
Pour tout élément de u de U faire :
– **Si** u qui n'appartient pas déjà à $Detat$, **alors** ajouter u à $Detat$ (sans le marquer) ;
– $Dtran[T, a] = u$ pour chaque lettre a dans l'alphabet associé au nœud p.

3.3 Minimisation de l'automate

Veillez à terminer une implantation du filtre `mgrep` avant d'implanter cette minimisation.

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 ci-dessous 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 en prenant $\Pi = \Pi_{new}$. Initialement, la partition Π comporte deux groupes, le groupe des états acceptants (finaux), et les états non-acceptants.

Algorithme de partitionnement d'une partition Π
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;

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.

4 Implantation de la commande `mgrep`

La syntaxe de la commande `mgrep` est la suivante :

```
mgrep  expression-rationnelle < fichier
```

en utilisant la grammaire des expressions rationnelles présentée dans la section 1. Cette commande effectue la recherche des motifs dans les lignes du fichier fourni sur son entrée standard ; elle affiche sur sa sortie standard les lignes qui contiennent un mot du langage défini par l'expression rationnelle.

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

```
% ./mgrep '(a|b)*abb' < foo
% ./mgrep \(a\|b\)\*abb < foo
```

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

5 Réalisation

Vous devez rendre (via l'interface PROF) une archive contenant :

- un fichier `Readme` ;
- les codes sources de votre projet ;
- un `Makefile` permettant la compilation de votre projet ;
- un fichier de test `mgrep-xampl.txt` et un script shell `mgrp-tst.sh` utilisant `mgrep` sur ce fichier test.

Vous pouvez (!) adjoindre une description de votre implantation.