ANALYSE LEXICALE

PHILIPPE LANGEVIN

TABLE DES MATIÈRES

1. Liste chaînée 1
2. Table de symboles 2
3. La commande `flex` 3
4. Temps de calculs 4
5. Fonction de hachage 4
6. Hachage modulaire 5
7. Travaux dirigés 6
8. Travaux pratiques 7

L'analyse lexicale est la première phase d'une compilation de texte qui consiste à décortiquer une source de caractères pour déterminer une séquence de lexèmes : identificateurs, nombres et symboles spéciaux etc... Les identificateurs rencontrés au cours de l'analyse sont mémorisés dans une table de symboles utile aux autres phases du compilateur : analyse syntaxique et production de code. L'objectif de cette leçon est de construire un analyseur lexical qui gère une table de d'identificateurs dans une situation simplifiée. Nous supposerons que la source est un fichier de caractères contenant des lettres de l'alphabet, quelques symboles de ponctuations et l'espace pour séparer les mots. Il s'agit de lire le texte source en une passe, caractère par caractère, de former les lexèmes et de maintenir à jour une liste des mots rencontrés et de leur fréquence d'apparition. Notre problème suggère l'emploi des listes chai nées, dont nous présenterons une approche objet/méthode qui peut donner lieu à un exercice de programmation orientée objet. Les fonctions de hachages augmentent les performances de notre analyseur. Enfin, la mise en oeuvre des algorithmes est facilité par l'utilisation de la commande unix `flex`.

mots clefs : liste chaînée, table de symboles, analyse lexicale, fonction de hachage.

1. LISTE CHAÎNÉE

Une liste chaînée est une structure de données dynamique sur laquelle nous pouvons appliquer des requêtes et des opérations de modifications. Comme dans le cas des fichiers, un pointeur privé permet l'accès séquentiel à l'information. La figure (1) représente une liste de lettre. Supposons que pointeur d'accès à l'information PA pointe sur le maillon grisé. La lecture de l'information se fait par `Lire`(L) et l'affectation de l'information se fait par `Ecrire`(L, x) Les méthodes `predecesseur`(L) et `successeur`(L) déplace respectivement le pointeur vers le maillon précédent et suivant. La taille d'un objet liste chaînée est variable. On peut

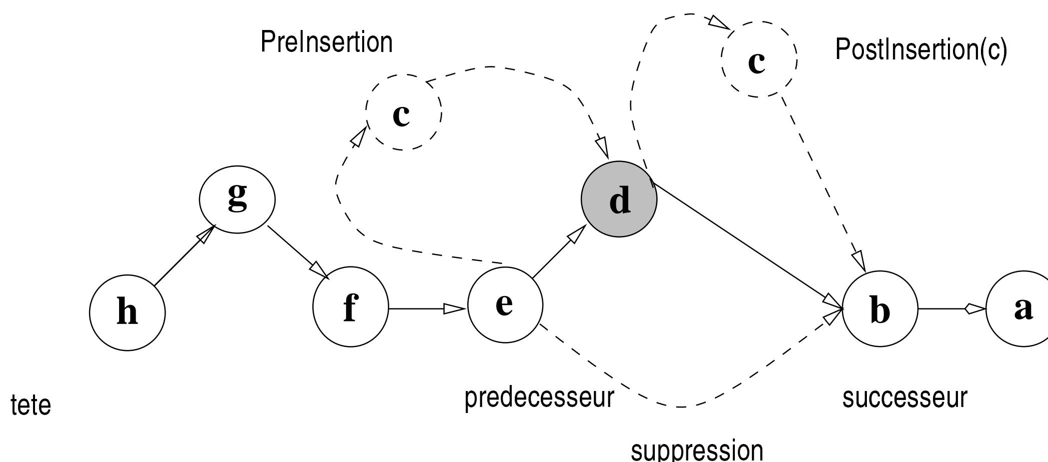

FIG. 1 – L'objet liste chaînée.

insérer et supprimer des des él'ements par les opérations d'insertions et de suppression. La méthode `PostInsertion`(c, L) ajoute un maillon immédiatement après PA. La méthode `PostInsertion`(c, L) fait un travail analogue. La méthode `suppression`(L) détruit le maillon pointé, le pointeur se déplace vers le suivant s'il existe et vers le précédentsinon. Trois fonctions logiques permettent de contrôler la liste : `vide`(L) qui renvoie vrai si la liste est vide, `fin`(L) retourne vrai si PA n'a pas de successeur et `debut`(L) renvoie vrai s'il n'a pas de prédécesseur. Enfin, la liste peut être rembobinée sur la tête ou sur la queue par `tete`(L) et `queue`(L).

Exercice 1. *Utiliser votre langage favori pour implanter l'objet liste chaînée et les méthodes associées.*

```
RECHERCHE( x, l)
DONNEES    x : MACHIN
           l : LISTE DE MACHIN
DEBUT
     SI VIDE(l) ALORS RETOURNER(0)
     POINTETETE(L);
     TANTQUE  NON FIN(L) ET ( x <> INFO(l) )
              SUCCESEUR(L)
     RETOURNER( NON FIN(l) )
FIN
```

2. TABLE DE SYMBOLES

L'analyse lexicale consiste en premier lieu à convenir de l'alphabet A des caractères sensés apparaître dans le fichier source. Dans notre exemple, A est réduit aux lettres et à l'espace. Nous devons aussi préciser la grammaire régulière qui définit le langage des lexèmes à reconnaître. Les caractères interdits sont ignorés et pourront déclencher des messages d'alertes.

```
LEXICAL ( texte )
```

```
        texte : FICHIER
        lst   : LISTE
        mot   : CHAINE
DEBUT
INITIALISER( lst )
TANTQUE NON FINFICHIER( texte )
        mot  =  LEXEME( texte )
        ENTETE( lst )
        TANTQUE  NON FINLISTE( lst ) ET ( MOT(lst) <> mot)
                 SUCCESSEUR(lst)
        SI FINLISTE(lst) ALORS POSTINSERTION(mot, lst)
        SINON    MISE-A-JOUR(lst)
FIN
```

On suppose une lecture séquentielle d'un fichier de caractères t. La procédure `LEXEME`(t) renvoie l'expression du lexème courant qui est recherché dans la liste l s'il est trouvé alors la fréquence est incrémentée par la procédure `MISEAJOUR`(l) sinon un nouveau maillon est insérer en fin de liste.

3. La commande `flex`

L'implantation de l'analyseur lexical est immédiate à partir de l'outil système `flex` (ex `lex`). Utiliser un copier/coller de la page du manuel de la commande `flex`, à partir des exemples proposés vous éditez un fichier `mots.lex` qui décrit le langage des identificateurs :

```
%{
int traitement( char* mot)
              {
              // A VOUS DE COMPLETER  !
              }
%}
ID       [a-z][a-z0-9]*
%%
{ID}     traitement(yytext);
 .       printf( "caracteres invalides : %s\n", yytext );
%%
int main( int argc, char** argv )
         {
         ++argv, --argc;
        if ( argc > 0 )
                     yyin = fopen( argv[0], "r" );
              else
                     yyin = stdin;

              yylex();
         }
```

La commande `flex` appliquée à `mots.lex` construit une source `C` qu'il suffit de compiler par `gcc -o mbots yy.lex.c -lfl`. Voilà, tout es dit, à vous de jouer !

4. Temps de calculs

Le temps de calculs de l'algorithme `Lexical`(t) dépend de la taille du fichier N mais aussi du nombre de mots μ distincts, à la louche le temps de calculs est $\mathrm{O}(N + \mu^2)$. Fixons, M un entier positif. Les mécanismes de hachage permettent de diviser par M le temps de calculs de notre algorithme. Donnons une fonction `Tiroir`(mot) qui associe au mot mot de A^* un entier de l'intervalle $[0, M[$ et modifions notre algorithme :

```
LEXICAL-HACHAGE ( texte )
             texte  : FICHIER
             table  : TABLEAU[m] de LISTE.
               mot  : CHAINE
                 i  : ENTIER
               lst  : LISTE
     DEBUT
     POUR i DANS [0, m[
            INITIALISER( table[i] )
     TANTQUE   ( NON FINFICHIER( texte) )
             mot =  LEXEME( texte )
             lst =  table[ TIROIR( mot )
             ENTETE( lst )
             TANTQUE  NON FINLISTE( lst ) ET ( MOT(lst) <> mot)
                      SUCCESSEUR(lst)
             SI FINLISTE(lst) ALORS POSTINSERTION(mot, lst)
             SINON    MISE-A-JOUR(lst)
     FIN
```

`LEXICAL-HACHAGE`($texte$) est est M fois plus performant que `LEXICAL`($texte$) dès que la fonction `TIROIR`(mot) répartit équitablement les mots de A^*, une telle fonction est une fonction de hachage.

5. Fonction de hachage

Comment construire des fonctions de hachage ? Remarquons que les mots de A^* peuvent être convertis en des nombres ce qui réduit la recherche des fonctions de hachages aux cas des entiers. Fixons M et N deux entiers naturels, une fonction de hachage d'un ensemble X de cardinal n dans l'intervalle $[0, M[$ doit satisfaire à une contrainte d'équirépartition :

$$(1) \qquad \forall x \in X, \quad \forall i \in [0, M[\quad , |\mathrm{Prob}\big(h(x) = i\big) - \frac{1}{M}| \leq \epsilon.$$

Les fonctions de hachages sont rares.

Exercice 2. *Paradoxe des anniversaires. La promotion 1999-2000 de maîtrise d'informatique est composée de* 25 *étudiants. Seriez-vous près à parier* 120 *Euros qu'au moins deux étudiants de cette promotion sont nés le même jour ?*

Essayons de nous convaincre de la rareté des fonctions de hachages. Supposons que M divise N et essayons d'évaluer le nombre de fonctions de hachage avec $\epsilon = 0$. Une telle fonction définit une partition de X en M parties de M éléments. Inversement, à une telle partition nous pouvons associer $M!$ fonctions mais une seule d'entre elles est une fonction de hachage...

6. HACHAGE MODULAIRE

Soit B un entier. Le hachage modulaire de base B applique les mots A^* dans l'intervalle $[0, M[$. Chaque lettres de A est codée par un entier, et le haché du mot $w = a_0 a_1 \ldots a8n$ est défini par :

$$h(w) = \sum_{i=0}^{n-1} \text{code}(a_i) B^i \mod M \tag{2}$$

Il s'agit donc dévaluer un polynôme dans l'anneau des entiers modulo M qu'on implante suivant le schéma d'évaluation de Hörner.

Exercice 3. *Ecrire le code C d'une fonction de hachage à M valeurs. Puis tester les performances de votre fonction de hachage en fonction de b et M.*

La qualité de la fonction de hachage modulaire varie fortement en fonction des paramètres. Pour B fixé, il faut éviter de prendre pour M les valeurs $B^k \pm r$, avec r petit.

Exercice 4. *Testez en fonctions de B et M la distribution des hachés d'un texte* aléatoire.

7. Travaux dirigés

Exercice 1. Donner une implantation en langage C de la structure de liste à partir des pointeurs structurés. Préciser les champs de cette structure puis écrire les primitives `FileVide`(F), `Enfiler`(x, F) et `Defiler`(F).

Exercice 2. Utiliser la structure de tableau pour implanter la structure de liste. Donner une solution dans laquelle toutes les primitives s'exécutent à temps constant.

Exercice 3. Donner une implantation des files à partir d'une pile.

Exercice 4. Écrire un algorithme `TriFusion`(L) pour trier une liste d'entiers dans l'ordre croissant. Quelle est la complexité de votre algorithme ? Préciser les avantages et les inconvénients de cet algorithme par rapport au tri fusion écrit dans le cadre des tableaux.

Exercice 5. Comment représenter les polynômes avec la structure de liste ? Quel est l'intérêt de cette approche ? Écrire les algorithmes d'additions et de multiplications de polynômes.

Exercice 6. L'algorithme `PARCOURS`(S, A, s) ci-dessous est une variante maladroite du parcours en largeur de graphe vu en cours. Prouver l'arrêt, et la correction de l'algorithme. Évaluer la complexité.

```
PARCOURS(S,A,s);
DONNEES
  S: ENSEMBLE;
  A: MATRICE;
  s: SOMMET;
VARIABLES
  x,y: SOMMET;
  v: TABLEAU;
DEBUT
  POUR CHAQUE x DANS S FAIRE
       v[x] = FAUX;
  FP
  ENFILER(s);
  TQ (NON-VIDE()) FAIRE
      x = DEFILER();
      v[x] = VRAI;
      POUR CHAQUE x DANS S FAIRE
         SI (NON v[y] ET A[x,y]) ALORS
              ENFILER(y);
         FSI
      FP
  FTQ
FIN
```

FIG. 2 – Un parcours maladroit

Exercice 7. Écrire un algorithme `Distribution`(t) pour déterminer la distribution des valeurs dans une table t. Estimer la complexité de votre algorithme.

8. Travaux pratiques

Durée : 1 séance de 3 heures.

Objectif : Écrire un programme pour calculer la distribution des mots dans un fichier texte. Il s'agit de gérer une liste de chaînes , ordonnée par ordre alphabétique et balisée par deux sentinelles. La commande unix flex sera utilisée pour extraire les mots du fichiers texte.

[1] Analyseur lexical

1. Éditer un fichier `lexical.lex` à partir de l'exemple.

2. Utiliser la commande `flex` pour produire le fichier `lexical.c`.

3. Compiler le programme en un exécutable `lexical.x`.

4. Tester ce programme.

5. Écrire un fichier `Makefile`.

[2] Insertion dans une liste ordonnée

6. Définir une structure pour représenter les listes chaînées de mots.

7. Écrire les procédures de gestions des listes ordonnées.

8. Modifier la source de `lexical.lex` pour obtenir un programme `distrib.x` calculant la distribution des mots d'un fichier texte.

9. Donner une information sur le temps de calcul de votre programme.

[3] Hachage modulaire

10. Écrire une fonction `int Hachage( char * mot, int B, int M)` qui calcule le haché d'un mot en base B modulo M.

11. Utiliser le principe du hachage pour accélérer le programme `distrib.x`. d'un facteur 100.

12. Tester vos programmes sur un gros fichier texte.

EXEMPLE DE PROGRAMME LEX

```
%{
#include <stdlib.h>
#include <stdio.h>

int cpt=0, mot=0, ligne=0, somme=0;
int ajoute(int s, char *str)
{
return( atoi(str)+s);
}
%}

CHIFFRE [0-9]
LETTRE  [A-Za-z]
NOMBRE  {CHIFFRE}+
MOT     {LETTRE}+
%%
{NOMBRE}  somme = ajoute(somme, yytext);
{MOT}   mot++;
\n      ligne++;
.       cpt++;
%%

int main(int argc, char *argv[])
{
printf("\nWelcome!\n");
++argv, --argc;
if ( argc > 0 )
          yyin = fopen( argv[0], "r" );
   else
          yyin = stdin;

yylex();
printf("\nNombre de mots lus :%d", mot);
printf("\nNombre de lignes   :%d", ligne);
printf("\ncaracteres ignores :%d", mot);
printf("\nsomme des nombres  :%d", somme);
printf("\nBye...\n");
}
```

MAKEFILE

```
LIB=-lfl
CC=gcc
LL=flex

prog.x: prog.lex
        $(LL) -oprog.c prog.lex
        $(CC) -o prog.x prog.c -lfl
```

Philippe Langevin, Octobre 2001.