Concours Centrale - Supélec 2003

Épreuve : INFORMATIQUE **Filière** MP

Association de parenthèses

Les candidats indiqueront clairement le langage de programmation choisi : Caml ou Pascal.

Ce problème concerne les mots bien parenthésés : il s'agit (dans la première partie) de déterminer si un mot donné est bien parenthésé (en un sens qui sera précisé) et, le cas échéant, de trouver la parenthèse fermante associée à une parenthèse ouvrante donnée (deuxième partie). La troisième partie donne une extension de ce problème.

Notes de programmation :

- *Pour Caml : on rappelle qu'une chaîne de caractère w de longueur n a ses lettres indexées de 0 à $n-1$; on a accès à celle d'indice k à l'aide de* `w.[k]`. *Par exemple, si* `w="informatique"`, *on a* `w.[0] = 'i'` *et* `w.[11] = 'e'`. *De même, un vecteur de taille n est indexé de 0 à $n-1$. Pour les questions utilisant des piles, les candidats ont à leur disposition une structure de pile caractérisée par :*
  - *un type polymorphe* `'a pile;`
  - *une fonction de création de pile vide* `creer_pile` *de signature*
    ```
    unit ->'a pile;
    ```
  - *une fonction* `empiler` *de signature* `'a ->'a pile -> unit` *permettant d'empiler un objet sur une pile;*
  - *une fonction* `dépiler` *de signature* `'a pile ->'a` *permettant d'extraire un objet d'une pile (le dernier empilé);*
  - *une fonction* `est_vide` *de signature* `'a pile -> bool` *testant si une pile est vide.*

  Bien entendu, les fonctions `empiler` *et* `dépiler` *sont à effet de bord : elles modifient la pile donnée en argument.*
- *Pour Pascal : pour les questions utilisant des piles, les candidats utiliseront une structure de pile d'entiers caractérisés par :*
  - *un type défini par* `: type pile;`
  - *une procédure de création de pile vide d'en-tête :*
    ```
    procedure creer_pile (var p:pile);
    ```

– *une procédure* `empiler` *permettant d'empiler un entier sur une pile ; son en-tête est :* `procedure empiler(v:integer;var p:pile);`
– *une fonction* `depiler` *permettant d'extraire un entier d'une pile (le dernier empilé); son en-tête est :*
```
function depiler(var p:pile):integer;
```
– *une fonction* `est_vide` *d'en-tête :*
```
function est_vide(p:pile):boolean;
```
testant si une pile est vide.

Bien entendu, la procédure `empiler` *et la fonction* `depiler` *sont à effet de bord : elles modifient la pile donnée en argument.*

Il est précisé qu'en Caml comme en Pascal, les candidats n'ont ni à définir le type `pile` ni à écrire les primitives associées données ci-dessus.

Partie I - Mots bien parenthésés

I.A - Le langage $\mathscr{L}_P$ des mots bien parenthésés

Dans cette partie, on s'intéresse au langage $\mathscr{L}_P$ des mots bien parenthésés. Initialement, il s'agit de mots sur l'alphabet constitué des deux lettres "(" et ")", c'est-à-dire les parenthèses ouvrante et fermante. Les mots de $\mathscr{L}_P$ correspondent à la suite des parenthèses pouvant intervenir dans une expression mathématique syntaxiquement correcte. Par exemple, l'expression

$$2+((x+y)*((x+2)*z))$$

nous fournit le mot bien parenthésé (()(())). Par contre, le mot)(n'est pas bien parenthésé, tout comme ()).

On donne comme définition : « un mot est "bien parenthésé" si à toute parenthèse ouvrante est associée une (unique) parenthèse fermante qui lui est postérieure ».

Dans cette première partie, il sera question d'expressions rationnelles. Les parenthèses apparaissant naturellement dans de telles expressions, la définition de $\mathscr{L}_P$ qui va suivre fait jouer à la lettre a le rôle de la parenthèse ouvrante "(" et à la lettre b le rôle de la parenthèse fermante ")" ; l'alphabet de travail est donc $\{a,b\}$.

On définit successivement des ensembles L_n $(n \in \mathbb{N})$ en posant $L_0 = \{\varepsilon\}$ (le langage constitué uniquement du mot vide) et, pour $(n \in \mathbb{N})$:

$$L_{n+1} = L_n \cup L_n^2 \cup aL_nb\,,$$

L_n^2 désignant l'ensemble de tous les concaténés possibles de deux mots quelconques de L_n, et aL_nb les mots de la forme $a \cdot w \cdot b$, pour w décrivant L_n.

Enfin, on définit : $\mathscr{L}_P = \bigcup_{n \in \mathbb{N}} L_n$.

Pour $w \in \{a,b\}^*$, $|w|$ désigne la longueur de w, et $|w|_a$ (resp. $|w|_b$ le nombre d'occurrences de la lettre a (resp. b) apparaissant dans w et, pour i entier, w_i désigne la i-ème lettre de w.

I.A.1) Montrer soigneusement que $abaabb \in \mathscr{L}_P$.

I.A.2) Montrer que pour tout $w \in \mathscr{L}_P$, on a $|w|_a = |w|_b$, et que si $w \neq \varepsilon$, alors il commence par un "a" et termine par un "b".

I.A.3) Montrer que lorsque $w \in \mathscr{L}_P$, alors pour tout i tel que $w_i = a$, il existe $j > i$ tel que $w[i..j] \in \mathscr{L}_P$, ou $w[i..j] = w_i \ldots w_j$ est le sous-mot de w constitué de ses lettres d'indices compris entre i et j. A-t-on unicité de j ?

I.B - Caractère non reconnaissable de $\mathscr{L}_P$

I.B.1) Montrer que le langage $\mathscr{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable par automate fini.

I.B.2) Justifier le fait que l'intersection de deux langages reconnaissables par automates finis est elle-même reconnaissable par un automate fini.

I.B.3) En décrivant $\mathscr{L}$ comme intersection de $\mathscr{L}_P$ et d'un langage bien choisi, montrer que $\mathscr{L}_P$ n'est pas reconnaissable par un automate fini.

I.C - Vérification du caractère "bien parenthésé"

I.C.1) Montrer soigneusement que $w \in \mathscr{L}_P$ si et seulement si $|w|_a = |w_b|$, et pour tout préfixe u de w, $|u|_a \geq |u|_b$.

I.C.2) Écrire une fonction `parenthese` déterminant si oui ou non un mot donné est bien parenthésé. On demande une fonction dont la complexité (en terme d'opérations élémentaires) soit linéaire en la taille de la chaîne fournie (avec une justification rapide).

Les candidats rédigeant en Caml écriront une fonction de signature :

```
parenthese : string -> bool=<fun>
```

et ceux qui rédigent en Pascal écriront une fonction d'en-tête :

```
function parenthese (w:string):boolean;
```

Partie II - Fermeture d'une parenthèse

Dans cette partie, on cherche à associer à chaque parenthèse ouvrante la parenthèse fermante associée. Pour des raisons de lisibilité, on reprend comme alphabet de travail l'ensemble constitué des lettres "(" et ")".

Par exemple, $(())()() \in \mathscr{L}_P$ alors que $())\notin \mathscr{L}_P$.

On a vu que lorsque $w \in \mathscr{L}_P$, alors pour tout i tel que $w_i = ($, il existe $j > i$ tel que $w[i\ldots j] \in \mathscr{L}_P$: si j est minimal, on dit que w_j est la fermante associée à l'ouvrante w_i, et réciproquement. Dans cette partie, l'objectif est d'obtenir toutes les associations ouvrantes/fermantes d'un mot bien parenthésé donné.

II.A - Deux algorithmes élémentaires

On propose ici deux algorithmes permettant de répondre au problème des associations : dans chaque cas, on demande de justifier et préciser l'algorithme, d'écrire une fonction (en Caml) ou une procédure (en Pascal) le mettant en œuvre, et de calculer la complexité de cet algorithme en terme d'opérations élémentaires, en fonction de la longueur du mot pris en entrée. En particulier, on exhibera des « cas limites » atteignant les complexités dans le pire des cas.

Notes de programmation :

- *En Caml : les fonctions auront pour signature :*

  ```
  associations:string -> int vect = <fun>
  ```

  et prendront en entrée une chaîne de caractère ***supposée bien parenthésée****. Si w a pour longueur n, on mettra en position i du vecteur résultat l'indice (compris entre 0 et $n-1$) de l'ouvrante/fermante associée à $w\cdot[i]$. Par exemple,*

  ```
  associations "(()())()";;
  ```

  retournera $[5;2;1;4;3;0;7;6]$.

- *En Pascal : on dispose d'un type* `tableau` *défini par*

  ```
  type tableau=array[1...1000] of integer;
  ```

  on écrira donc des procédures d'en-tête :

  ```
  procedure association(w:string; var resultat:tableau);
  ```

  ces procédures assigneront `resultat[i]` *pour* $1 \leq i \leq |w|$. *Par exemple, l'exécution de*

  ```
  association ("(()())()",t) ;
  ```

  affectera aux 8 premières cases du tableau t les valeurs $6, 3, 2, 5, 4, 1, 8, 7$.

II.A.1) **Premier algorithme** : pour chaque i tel que $1 \leq i \leq |w|$, on cherche le plus petit $j > i$ tel que $w[i..j]$ est bien parenthésé.

II.A.2) **Second algorithme** : on utilise une structure de pile (LIFO : on dépile le dernier objet empilé). On part d'une pile vide et on lit les lettres successives de $w \in \mathscr{L}_P$: lorsqu'on rencontre une parenthèse ouvrante $w_i = ($, on empile sa position i, et lorsqu'on rencontre une parenthèse fermante $w_j =)$, on dépile un entier k de la pile, et la parenthèse ouvrante associée à w_j est alors w_k.

II.B - Utilisation de l'arbre réduit

Deux mots $w_1, w_2 \in \{(,)\}^*$ sont dits *équivalents* (et on note $w_1 \sim w_2$) lorsqu'on peut obtenir l'un à partir de l'autre en faisant apparaître et disparaître des facteurs $()$. Par exemple : $(() ())) \sim ((()))) \sim ()))) \sim))) \sim)) () \sim \ldots$

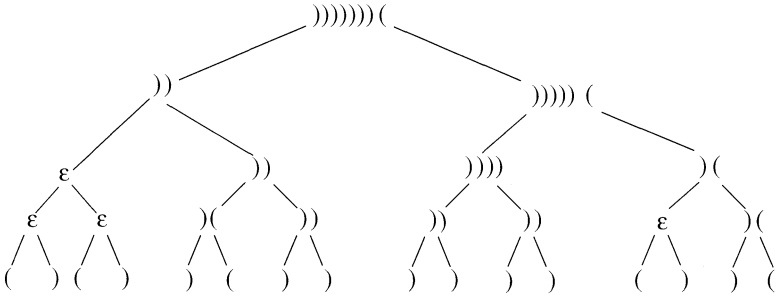
Un mot de $\{(,)\}^*$ sera dit *irréductible* s'il n'est équivalent à aucun mot de longueur strictement plus petite.

II.B.1) Montrer que tout mot w de $\{(,)\}^*$ est équivalent à un unique mot irréductible w', et que celui-ci est de la forme $)^i(^j$, avec $i, j \in \mathbb{N}$. *On dit alors que w' est le représentant irréductible de w.*

II.B.2) En supposant w_1 et w_2 irréductibles, comment calculer la forme irréductible de leur concaténé $w_1 w_2$?

II.B.3) Comment *caractériser* les mots bien parenthésés en terme de représentant irréductible ? Justifier le résultat énoncé.

*Soit $w \in \{(,)\}^*$ de longueur 2^p (pour simplifier l'exposé ; **on maintient d'ailleurs cette hypothèse dans la suite de cette partie**). On définit* l'arbre réduit de w *par la construction suivante : c'est un arbre binaire complet de hauteur p dont les nœuds sont étiquetés par des mots de $\{(,)\}^*$. Les 2^p feuilles sont étiquetées par les lettres de w, et chaque nœud interne a pour étiquette le représentant irréductible du concaténé de ses deux fils (cet arbre se remplit donc « étage par étage », des feuilles vers la racine) :*



Exemple : arbre réduit de $\omega_0 = ()())((((((())()($

II.B.4) Évaluer (en fonction de $|w|$) le temps nécessaire pour construire l'arbre réduit d'un mot $w \in \{(\ ,)\}^*$.

II.B.5) Construire l'arbre réduit de $w_1 = ((((()(()))))))()$, puis montrer que $w_1 \in \mathscr{L}_P$.

II.B.6) Proposer un algorithme permettant de calculer la parenthèse fermante associée à une ouvrante donnée de $w \in \mathscr{L}_P$ en temps $O(\ln|w|)$, une fois l'arbre réduit de w construit. On demande impérativement de le mettre en œuvre dans la recherche de la fermante associée à la quatrième ouvrante du mot w_1 défini plus haut.

II.B.7) Pour obtenir l'ensemble des associations ouvrante/fermante d'un mot bien parenthésé, donner une évaluation du temps nécessaire si on utilise l'arbre réduit de w. Comparer avec les algorithmes élémentaires précédents.

II.B.8) On suppose qu'on dispose d'une grande quantité de processeurs (de l'ordre de $|w|$) capables de travailler en même temps. Expliquer informellement comment construire l'arbre réduit de $|w|$ en temps $O(\ln|w|)$ avec $O(|w|)$ processeurs.

Partie III - Parenthésage hétérogène (ou colorié)

On s'intéresse ici aux parenthésages hétérogènes, ou coloriés, pour lesquels on s'autorise différents types de « parenthésage », par exemple avec des parenthèses et crochets : on peut imbriquer différents parenthésages, sans les croiser : ([ ]) est bien parenthésé mais pas ([)]. De même, ([[()]()[ ]]()) est bien parenthésé mais pas [([[()]()[ ]]][ ])

III.A - Proposer une définition formelle du langage $\mathscr{L}'_P$ des mots bien parenthésés avec les deux types de parenthèses précédents ; l'alphabet est constitué de quatre lettres : $A = \{(\ ,),[\ ,]\}$.

III.B - $\mathscr{L}'_P$ est-il rationnel ? Écrire une fonction prenant en entrée une chaîne de caractère w sur l'alphabet A et retournant `true` ou `false` selon que w est dans $\mathscr{L}'_P$ ou non.

III.C - Écrire une fonction (en Caml) ou une procédure (en Pascal) nommée `associations` prenant en entrée une chaîne de caractère **que l'on suppose bien parenthésée** et calculant un vecteur (en Caml) ou un tableau (en Pascal) contenant les associations ouvrantes/fermantes avec les conventions de numérotation précisées dans la partie II du problème.

Par exemple, si l'entrée est ([ ]([ ]))[ ], *la liste retournée en Caml sera* [7;2;1;6;5;4;3;0;9;8] *et le tableau calculé en Pascal aura pour premières valeurs* : 8, 3, 2, 7, 6, 5, 4, 1, 10, 9 .

Évaluer la complexité de ce programme.

••• FIN •••