UTBM le 12 Novembre 2001

MEDIAN – LO42

Les documents ne sont pas autorisés (La copie ou les idées du voisin non plus).
Toute réponse devra être claire et justifiée si nécessaire (toute ambiguïté sera mal interprétée).

1) Analyseurs prédictifs (5)

Soit la grammaire simplifiée des expressions arithmétiques:

E	→	T E'		
E'	→	+ T E'	|	ε
T	→	F T'		
T'	→	* F T'	|	ε
F	→	(E)	|	id

Cette grammaire est sans récursivité à gauche (et factorisée à gauche).

Sur la base de cette grammaire on peut écrire un analyseur syntaxique prédictif. La méthode la plus courante employée pour l'analyse syntaxique d'un texte du langage défini, est une descente récursive sans rebroussement. La construction d'un tel analyseur nécessite, pour un non terminal A, une alternative de production **unique** sur le symbole courant a. Ceci signifie que des règles de la forme ci-dessous ne peuvent être exploitées pour une analyse descendante (nécessité de factorisation à gauche).

A → a A0 | b A0 | a A1

Il est possible de créer un analyseur prédictif par descente récursive, On élabore les programmes en respectant la démarche suivante :

- pour chaque non terminal A, créer une procédure;
- pour chaque production $\mathbf{A \rightarrow X_1 X_2 \ldots X_n}$ créer un chemin de reconnaissance de l'expression.

La concaténation sera exprimée par la séquence.
Chaque fois que l'on rencontre un non terminal on appellera la procédure de reconnaissance de ce non terminal.
Lorsque l'on rencontre un terminal il est comparé avec ceux prévus dans la grammaire. Selon le cas il sera accepté ou provoquera une erreur.

L'analyseur à descente récursif peut être :

```
E()
Début
    T(); Ep();
Fin ;

Ep()
Début
    si (symbole = '+') alors accepter('+'); T(); Ep();
    sinon ;
Fin;

T()
Début
    F(); Tp();
Fin

Tp()
Début
    si (symbole = '*') alors accepter('*'); F(); Tp();
    sinon ;
Fin;


F()
Début
```

```
        si (symbole = '(') alors accepter('('); E(); accepter (')');
        sinon   si (symbole = id)  alors accepter(id);
                sinon erreur();
Fin;
```

Les fonctions annexes peuvent être :

```
unit symbole;
main()
Début
    symbole ← Anal_lex();
    E();
Fin;

accepter(lex)
unit lex;
Début
    si (symbole = lex) alors symbole ← Anal_lex();
    sinon erreur();
Fin;

erreur()
Début
    ecrire("Erreur de syntaxe\n");
    arret_programme(1);
Fin;
```

où la fonction Anal_lex() est une fonction qui scrute le texte en entrée est réalise l'analyse lexicale. Elle retourne un élément appelé « token» ou « unité lexicale» correspondant à un symbole ou mot réservé de la grammaire ou un ensemble de mots comme 'id' représente tous les identificateurs possibles.

1 On vous demande de supprimer La fonction Ep() en éliminant la récursivité puis en l'intégrant dans E()

On vous demande de réduire les appels récursifs en éliminant les procédures Ep() et Tp():

Solution

```
Ep()
{   int boucle = vrai;
    while(boucle) {
        if (symbole == '+') accepter('+'); T();
        else boucle = faux;
    }
}

E()
{   int boucle = vrai;
    while(boucle) {
        T();
        if (symbole == '+') accepter('+');
        else boucle = faux;
    }
}
```

de même pour T() et Tp() :

```
T()
{   int boucle = 1;
    while(boucle) {
        F();
        if (symbole == '*') accepter('*');
        else boucle = 0;
    }
}
```

2) Structure récursive construite récursivement

Sur la base d'une structure récursive classique nous avons défini une fonction récursive « jolie_f » de manipulation et une fonction d'accès.

```
Type    liste : ^elt_liste ;
        elt_liste : structure debut
                        Info : elt ;
                        Suivant : liste ;
                fin ;

Fonction jolie_f() : liste ;
Var     val : elt ;
        fin : booleen ;
        aux : liste ;
Début
        lire (valeur, fin) ;
        si fin alors
                jolie_f ← nul ;
        sinon
                nouveau (aux) ;
                aux^.info ← valeur ;
                aux^.suivant ← jolie_f () ;
                jolie_f ← aux ;
        fsi
Fin ;
```

Où **lire(valeur, fin)** est une procédure qui permet de placer dans le paramètre « valeur » une valeur à traiter et de positionner le booléen « fin » à faux. Le booléen « fin » recevra vrai s'il n'y a plus de valeur à traiter.

```
Fonction accès(x : liste) : Telt ;
Début
        si x <> nul alors
                accès ← x^.info
        sinon
                accès ← -1 ;
        fsi ;
fin ;
```

1) On vous demande de préciser l'action de la fonction et d'indiquer à quel TDA correspond cette opération en relation avec l'ordre d'accès.

2) On désire un TDA qui inverse l'accès par rapport à celui défini. Ecrivez une **fonction récursive** qui permet de définir la même opération que « jolie-f » pour ce nouveau TDA (La signature de la fonction peut changer). L'opération d'accès devra rester inchangée.

Solution

1) La fonction permet de construire **une liste chaînée**.
   La fonction construit une liste chaînée mais l'utilisation de l'association des deux fonctions apporte une information supplémentaire à savoir l'ordre d'utilisation des éléments de la suite. L'accès étant sur le premier de la liste, c'est à dire la première valeur lue, nous sommes en présence d'un fonctionnement « Premier arrivé Premier servi », d'où le TDA **file**.

2) Nous utiliserons une fonction avec un paramètre.

```
Fonction const_pile ( p : pile) : pile ;
Var val : elt ;
        fin : booleen ;
        aux : pile ;
Début
    lire (valeur, fin) ;
    si fin alors
                const_pile ← p ;
        sinon       nouveau (aux) ;
```

```
            aux^.info ← valeur ;
            aux^.suivant ← p ;
            const_pile ← const_pile(aux ) ;
        fsi
    Fin ;
```

3) Les jolies couleurs

Nous disposons d'un tableau contenant des éléments de trois couleurs différentes (rouge, blanc, bleu). Nous souhaitons grouper les couleurs selon l'ordre suivant (bleu | blanc | rouge).
Pour ce faire vous ne pouvez pas utiliser un autre tableau. Il vous faut faire le tri sur place donc dans le tableau initial.

1. définir l'invariant , c'est à dire la propriété qui défini l'état du tableau après le $i^{ème}$ traitement.
2. Donner en pseudo-code une solution à ce problème.

Solution

1) La particularité du tri demandé est que le type couleur ne possède pas à priori pas de relation d'ordre. D'autre part deux éléments d'une même couleur peuvent être à des places quelconques dans la plage de couleur ce qui signifie qu'il est inutile de les comparer entre eux. Nous avons vu quelque chose d'approchant lorsque nous avons développé le tri Quicksort avec la procédure « placer » où nous construisions deux plages : les éléments supérieurs à la valeur test et les éléments inférieurs à cette valeur.

Au cours de notre tri nous créerons trois plages de couleur. A l'étape i, c'est à dire après le traitement de l'élément i, nous aurons les élément de 1 à i tris en trois plage et les éléments de i+1 à n non triés.

Exemple :

Bl	Bl	Bl	B	B	B	R	R	R	R	R	R	B	Bl	Bl	B	B	R	R	B	B	R	Bl	B	R
			k			j						i												

D'où l'invariant

$\{1 \le k \le j \le i \le n\} \wedge \{l \in [1,k[, t[l]=bleu\} \wedge \{m \in [k,j[, t[m]=blanc\} \wedge \{o \in [j,i[, t[o]=rouge\}$

On pourra écrire un algorithme basé sur cet invariant. Dans cet algorithme, pour positionner un élément bleu on devra décaler la plage des blancs ainsi que celle des rouges. De même pour placer un blanc on devra décaler la plage des rouges. Si l'élément à traiter est rouge il suffira de le laisser sur place.
Pour décaler une plage de couleur on peut procéder par décalages successifs de tous les éléments de la plage. Cette solution n'est pas satisfaisante du tout ! en effet, les éléments d'une même couleur étant indifférenciés pour décaler la plage sur la droite il suffit de prendre l'élément le plus à gauche et de le déposer sur la droite de la plage.
En outre, mettre en œuvre la solution sous cette forme impose un décalage de deux plages pour placer un élément de couleur bleu. Pour éviter ceci nous placerons les rouges là où il doivent être définitivement. C'est à dire que la plage rouge sera placée directement en fin de tableau. (voir similitude avec la procédure placer)

Exemple :

Bl	Bl	Bl	B	B	B	R	B	Bl	Bl	B	B	R	R	B	B	R	Bl	B	R	R	R	R	R	R
			k			i												j						

D'où l'invariant

$\{1 \le k \le i \le j \le n\} \wedge \{l \in [1,k[, t[l]=bleu\} \wedge \{m \in [k,i[, t[m]=blanc\} \wedge \{o \in]j,n], t[o]=rouge\}$

Le traitement sera alors :

```
Selon t[i]
 cas rouge :
        Temp ← t[i] ;
        t[i] ← t[j] ;
        t[j] ← temp ;
        j ← j-1 ;
 cas bleu :
        k ← k+1 ;
        temp ← t[i] ;
        t[i] ← t[k] ;
        t[k] ← temp ;
        i ← i + 1 ;
 autre :
        i ← i + 1 ;
finselon;
```

La boucle sera :

```
tantque i ≤ j faire
        traitement(t[i])
fait
```

En début de traitement les plages seront vides d'où les initialisations

```
i ←1 ;   { élément à traiter }
k ←1 ; { 1ère position à droite de la plage bleu }
j ← n ; { 1ère position à gauche de la plage rouge }
```

d'où l'ensemble

```
i ←1 ;   { élément à traiter }
k ←1 ; { 1ère position à droite de la plage bleu }
j ← n ; { 1ère position à gauche de la plage rouge }
tantque i ≤ j faire
        Selon t[i]
         cas rouge :
                Temp ← t[i] ;
                t[i] ← t[j] ;
                t[j] ← temp ;
                j ← j-1 ;
         cas bleu :
                k ← k+1 ;
                temp ← t[i] ;
                t[i] ← t[k] ;
                t[k] ← temp ;
                i ← i + 1 ;
         autre :
                i ← i + 1 ;
        finselon;
fait
```

Remarque : ici si l'invariant est correct nous ne pouvons pas compter les étapes comme précédemment. En effet, si nous sommes devant le traitement de l'élément d'indice i nous sommes à l'étape « i+n-j » du fait du positionnement des rouges en fin de tableau.

Dans le cas où le déplacement d'un élément serait une opération « lourde » par rapport au test de couleur, on peut vouloir éviter la permutation du rouge avec un bleu ou un rouge que l'on devra déplacer.

```
        Selon t[i]
```

```
cas rouge :
        Tantque t[j] = rouge et j>i faire j ← j - 1  fait ;
        Si i<>j alors
                temp ← t[i] ;
                Si t[j]= bleu alors
                        k ← k+1 ;
                        t[i] ← t[k] ;
                        t[k] ← t[j] ;
                sinon
                        t[i] ← t[j] ;
                Fsi ;
                t[j] ← temp ;
                j ← j - 1 ;
        Fsi ;
cas bleu :
                k ← k+1 ;
                temp ← t[i] ;
                t[i] ← t[k] ;
                t[k] ← temp ;
finselon
i ← i + 1 ;
```

(Ici on teste deux fois l'élément que l'on permutera avec le rouge: dans le tantque et dans le test bleu. Il est toutefois possible d'utiliser un booléen pour éviter ce double test et construire une boucle globale.)