

I. La réversivité

1. Principe et types de réversivité

- **Réversivité** : Une entité est réversive lorsqu'on l'utilise pour la définir. Une fonction réversive s'« auto-appelle ».
 - **Réversivité terminale** : l'appel réversif est la dernière instruction et est isolé.
 - **Réversivité non terminale** : l'appel réversif n'est pas la dernière instruction ou est dans une expression.

2. Méthode d'écriture d'un programme réversif

- Identifier le ou les cas particulier
- Identifier le cas général qui effectue la réversion

Lors de l'appel réversif, on est utilisateur donc on considère que le programme appelé fonctionne.

II. Structure dynamique de données

1. Notions de création et de fonctionnement d'un programme

a. La compilation

La compilation permet de transformer du code humainement compréhensible vers du code machine. De plus, il permet souvent de changer de paradigme et peut rajouter du code.

b. Segments mémoire et allocation

Les entités utilisées sont placées dans la mémoire vive dans divers segments :

- **statique ou *text*** : programmes et sous-programmes
- ***bss*** : variables globales
- ***data*** : constantes
- ***tas* ou *heap*** : espaces alloués dynamiquement
- ***pile* ou *stack*** : espaces alloués statiquement
- **Allocation statique** : Allocation prévue à la compilation (variables locales, paramètres, ...)
- **Allocation dynamique** : Allocation non prévue à la compilation, écrite par le programmeur.

2. Les pointeurs

Un *pointeur* **p** est une variable de type « *pointeur sur T* » noté **^T** référant une zone mémoire permettant de stocker une information de type **T**. Un pointeur à **NIL** ne pointe sur rien.

p^ permet d'accéder à l'espace mémoire pointé par **p**. **@var** permet d'obtenir un pointeur vers la variable **var**.

Exemple :

p: ^Entier
i: Entier

p ← NIL
p ← @i
p^ ← 3

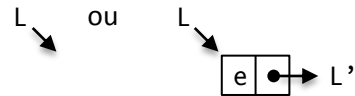
3. Allocation dynamique

p : ^T	Pseudo-code	Pascal
Allouer p	allouer(p)	new(p)
Libérer p	liberer(p)	dispose(p)

4. Les listes chaînées

a. Définition

Une liste chaînée est soit une liste vide, soit un élément suivi d'une liste chaînée (un nœud).



b. Fonctions de base

Type ListeChaîneeDEntiers = ^NoeudDEntier

Type NoeudDEntier = **Structure**

entier : Entier

listeSuivante : ListeChaîneeDEntiers

finstructure

- **fonction** listeVide () : ListeChaînee
{ retourne une liste vide }
- **fonction** estVide (liste : ListeChaînee) : Booleen
{ indique si une liste est vide ou non }
- **procédure** ajouter (E/S liste : ListeChaînee, E element : Entier)
{ ajoute un nœud en tête de liste }
- **fonction** obtenirEntier (liste : ListeChaînee) : Entier
 - Précondition non(estVide(liste)){ retourne l'entier du nœud donné }
- **fonction** obtenirListeSuivante (liste : ListeChaînee) : ListeChaînee
 - Précondition non(estVide(liste)){ retourne la liste sans son élément de tête }
- **procédure** fixerListeSuivante (E/S liste : ListeChaînee, E nelleSuite : ListeChaînee)
 - Précondition non(estVide(liste)){ change la suite de la liste après le nœud donné }
- **procédure** supprimerTete (E/S liste : ListeChaînee)
 - Précondition non estVide(liste){ supprime l'élément en tête de liste }
- **procédure** supprimer (E/S liste : ListeChaînee)
{ supprime l'intégralité de la liste }

c. Conception détaillées de quelques fonctions

procédure ajouter (E/S l : ListeChaînee, E e : Entier)

Déclaration temp : ListeChaînee

debut

temp ← l

allouer(l)

l^.entier ← e

fixerListeSuivante(l,temp)

fin

procédure supprimerTete (E/S l : ListeChaînee)

Précondition non estVide(l)

Déclaration temp : ListeChaînee

debut

temp ← l

l ← obtenirListeSuivante(l)

liberer(temp)

fin

procédure supprimer (E/S l : ListeChaînee)

debut

tant que non estVide(l) **faire**

supprimerTete(l)

fin tant que

fin