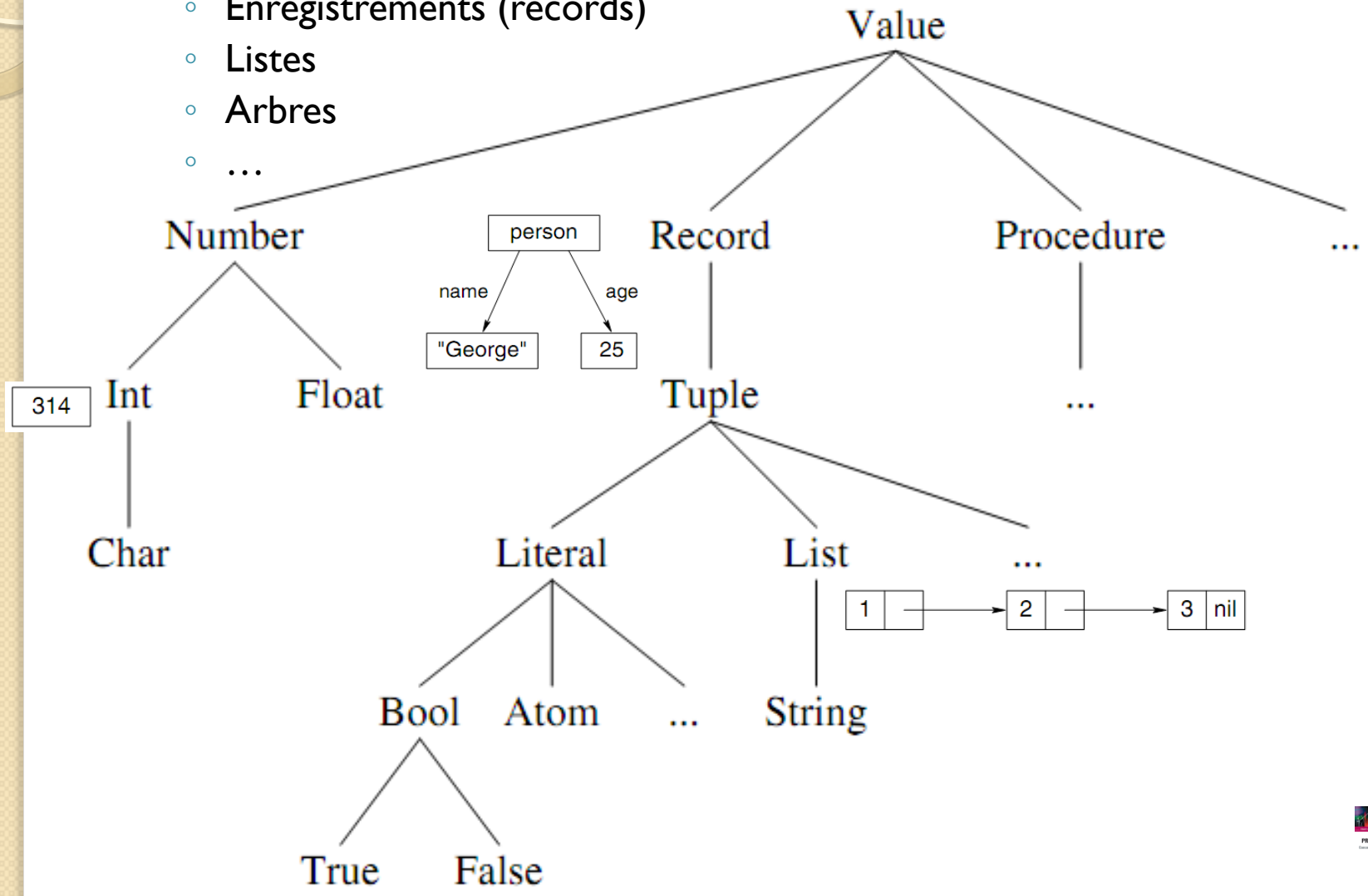
[Concepts de programmation]

pptPlex Section Divider

The slides after this divider will be grouped into a section and given the label you type above. Feel free to move this slide to any position in the deck.

Valeurs

- Constantes numériques et symboliques
- Entités contenant des constantes
  - Enregistrements (records)
  - Listes
  - Arbres
  - …



Mémoire et variables

- Mémoire à affectation unique
  - Ensemble de variables déclaratives
  - Une variable déclarative
    - Un nom pouvant être lié à une seule valeur (initialement aucune)
    - Une « cas mémoire » pouvant être remplie une seule fois (initialement vide
  - Liaison d'une valeur à une variable
    - La variable devient un autre nom de la valeur
    - Rien ne distingue variable et valeur
- Mémoire à valeurs
  - Toutes les variables sont toujours liées à une seule valeur (pas de variable non liée).
- Mémoire à affectation multiple
  - Ensemble de variables déclaratives et de cellules
  - Une cellule
    - Un nom associé à un contenu qui peut être modifié (affecté) plusieurs fois
    - Une « case mémoire » dont le contenu peut être affecté plusieurs fois
  - Affecter = modifier le contenu d'une variable

Identificateur

- Nom d'une variable utilisée dans et par un programme
- Quand on raisonne sur les variables, on manipule
  - Les noms abstraits que l'on donne aux variables
  - Les identificateurs

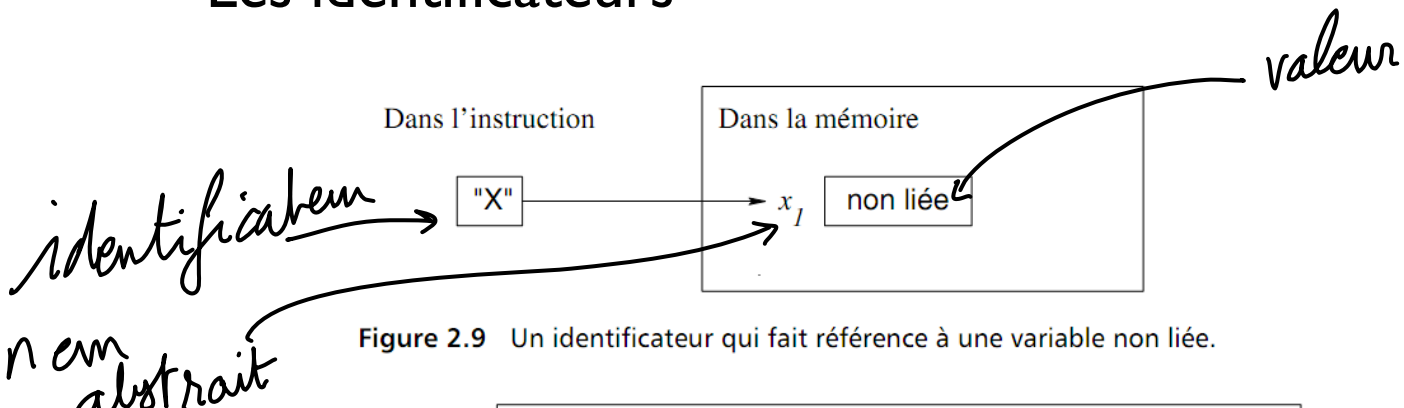


Figure 2.9 Un identificateur qui fait référence à une variable non liée.

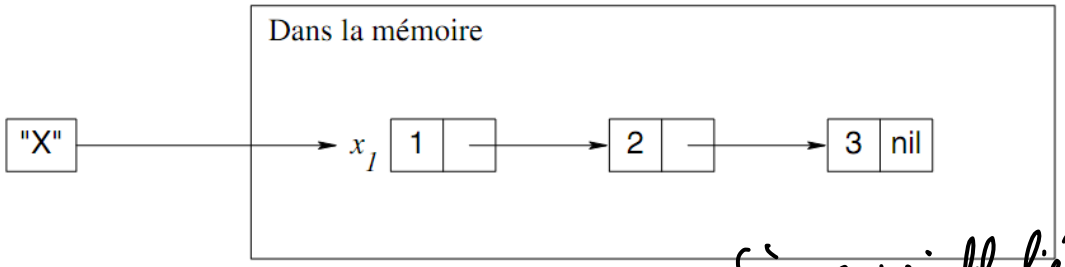
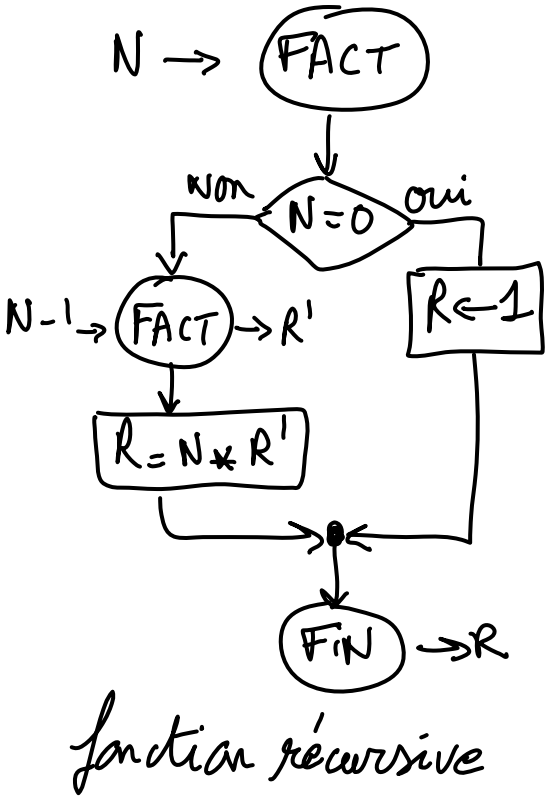


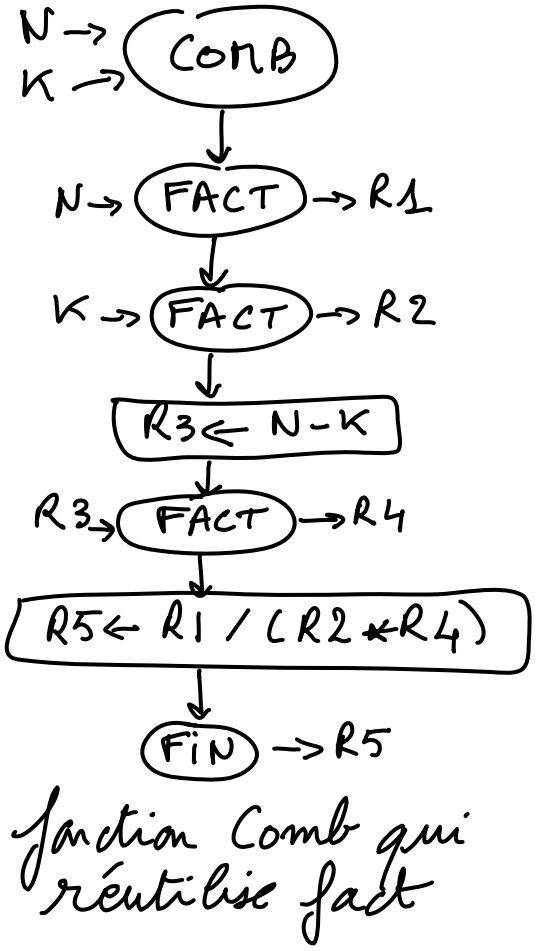
Figure 2.11 Un identificateur qui fait référence à une valeur.

Procédures (et fonctions)

- Sous-programme
  ensemble d'instructions formant un tout cohérent pouvant être exécuté indépendamment
- Propriétés
  - Récursivité
  - Abstraction fonctionnelle
    - Utiliser des fonctions
      pour définir d'autres fonctions
    - Programmation d'ordre supérieure

Procédures (et fonctions)





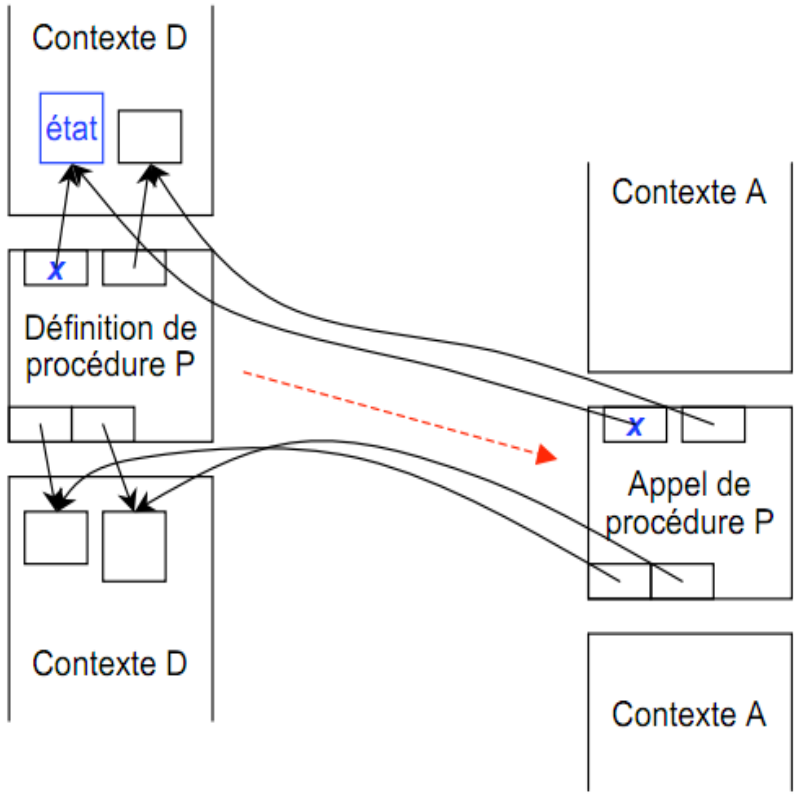

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

La fermeture

- Fermeture à portée lexicale (“lexically scoped closure”)
- Permet les procédures
- Point de vue implémentation
  - une procédure
  - avec ses références externes
- Point de vue utilisateur,
  - “paquet de travail”
  - on peut créer un paquet **comme une valeur** (une constante) à un endroit du programme, le donner à un autre endroit et décider de l’exécuter à cet endroit. Le résultat de son exécution est le même celui obtenu par exécution directe.
- Permet (avec la récursivité) de programmer toute structure de contrôle: boucles, conditionnels, etc.

La fermeture

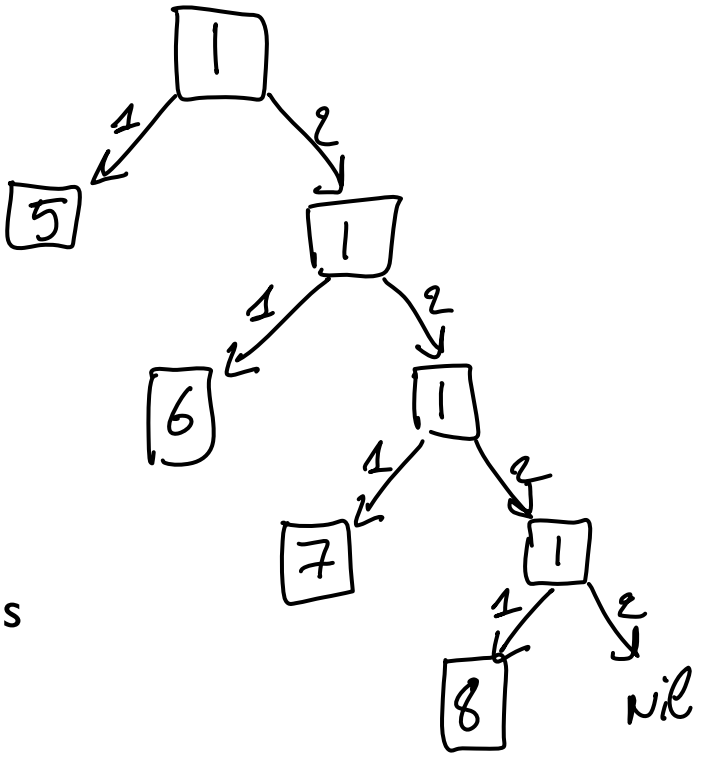


- La définition de P créé une fermeture: elle se souvient du contexte D de sa définition
- À l'appel de P, la fermeture utilise le contexte D
- Un objet est un exemple d'une fermeture: ***x*** fait référence à l'état de l'objet (un de ses attributs)

Les enregistrements

- Regroupement de données avec un accès direct à chaque donnée
  - R=chanson(nom:"Erlkönig" artiste:"Dietrich Fischer-Dieskau" compositeur:"Schubert")
  - R.nom est égal à "Erlkönig"
    - chanson = étiquette
    - compositeur:Schubert = champ de nom (« feature ») compositeur
- L'enregistrement est à la base de la programmation symbolique
  - Calculer avec des enregistrements: les créer, les décomposer, les examiner, et ce pendant l'exécution
  - ➔ les listes, les chaînes et les arbres
  - Nécessaires pour
    - la programmation orientée objet
    - les interfaces graphiques
    - la programmation par composants

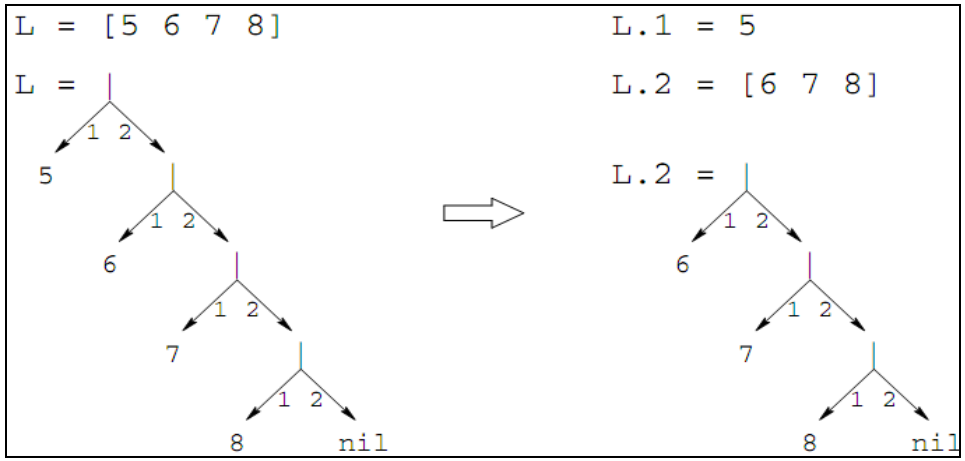
Les listes



- Séquence d'éléments délimitée par des crochets e.g. [5,6,7,8]
- Liste vide : nil
- Séquence d'enregistrements
  - D'étiquette « | »
  - Et dont les champs ont les noms (« features ») 1 et 2
  -
```
|(1:5 2:
        (|(1:6 2:
                (|(1:7 2:
                        (|(1:8 2:
                                (nil)
                        )
                )
        )
 )
```
  - Raccourci : `5|(6|(7|(8|nil)))`

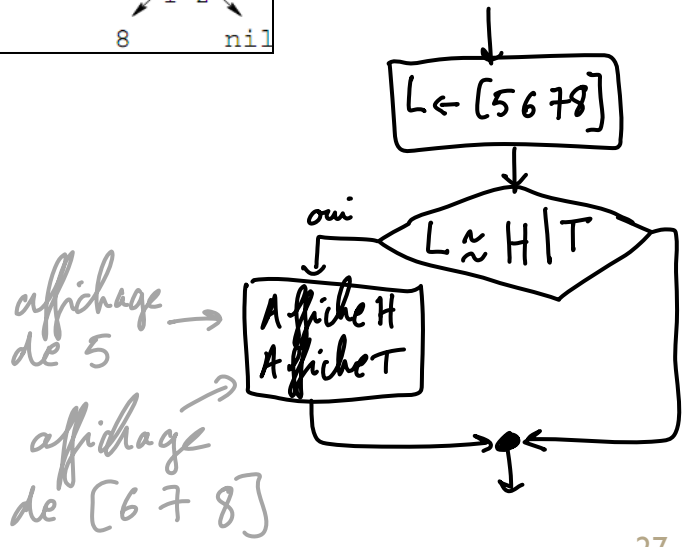
Manipulation de listes

- « . » pour l'accès aux champs



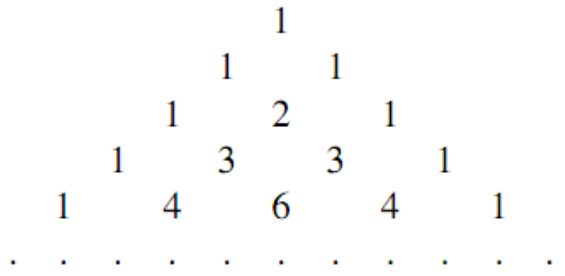
- case … of …
  pour séparer la tête
  de la queue

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```



La programmation d'ordre supérieur

- Capacité de passer une fonction comme argument d'une autre fonction
- Ex. : le triangle de Pascal
  - Chaque ligne stockée dans une liste
  - Génération de la ligne sous `[1 3 3 1]` par




```
  [1  3  3  1  0]
+ [0  1  3  3  1]
-----------------
  1  4  6  4  1
```

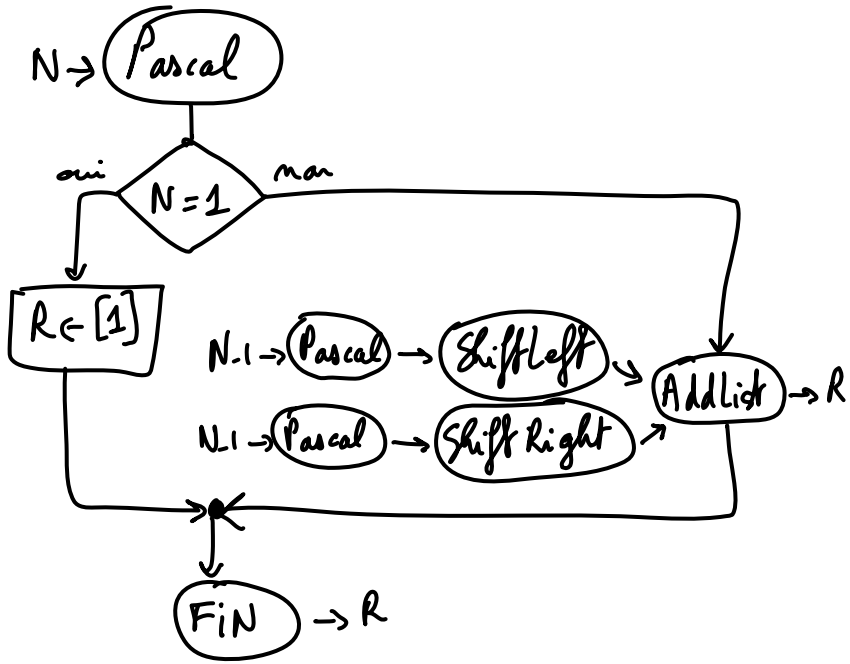
Triangle de Pascal, version de base

```
declare Pascal AddList ShiftLeft ShiftRight
fun {Pascal N}
   if N==1 then [1] else
      {AddList {ShiftLeft {Pascal N-1}}
               {ShiftRight {Pascal N-1}}} end
end
```

```
fun {ShiftLeft L}
   case L of H|T then H|{ShiftLeft T} else [0] end
end
```

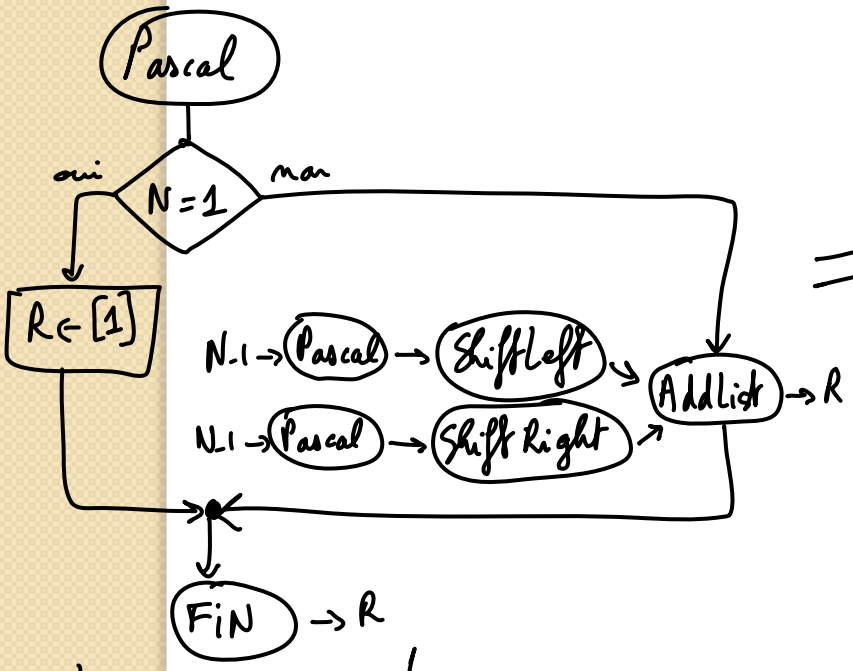
```
fun {ShiftRight L} 0|L end
```

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then H1+H2|{AddList T1 T2} end
   else nil end
end
```

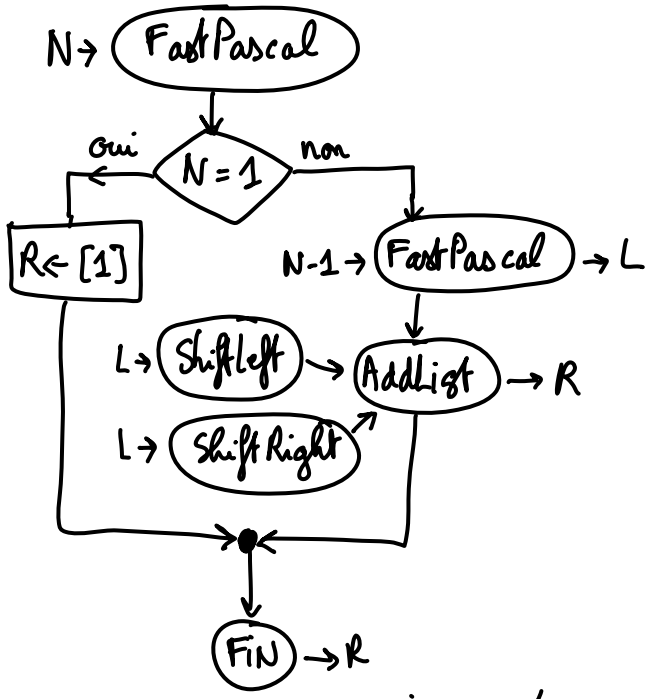


Triangle de Pascal, version optimisée

```
declare
fun {FastPascal N}
   if N==1 then [1] else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}} end
end
```



Version de base

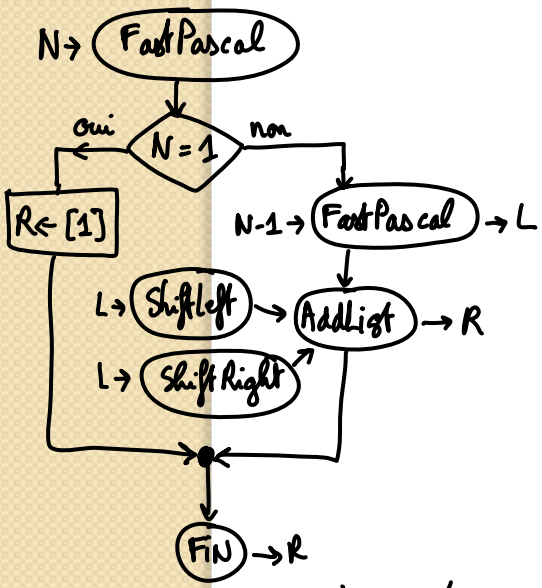


Version optimisée

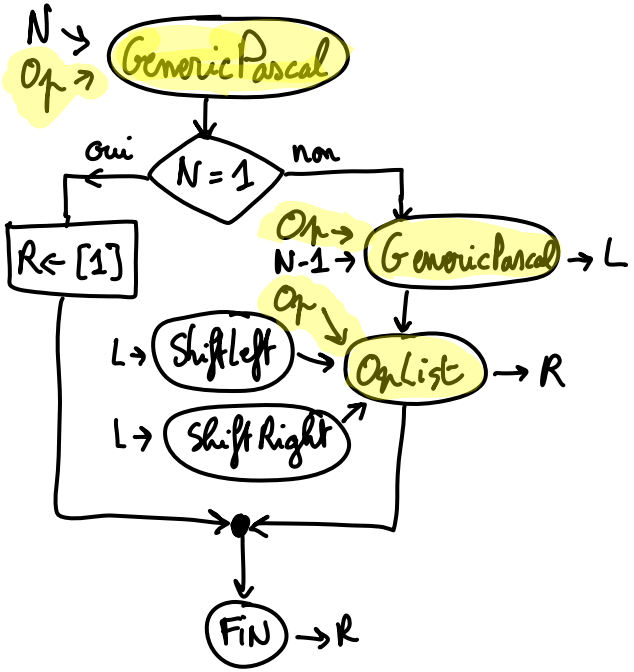
Triangle de Pascal, version générique

```
declare GenericPascal OpList
fun {GenericPascal Op N}
   if N==1 then [1] else L in
      L={GenericPascal Op N-1}
      {OpList Op {ShiftLeft L} {ShiftRight L}}
   end
end
```

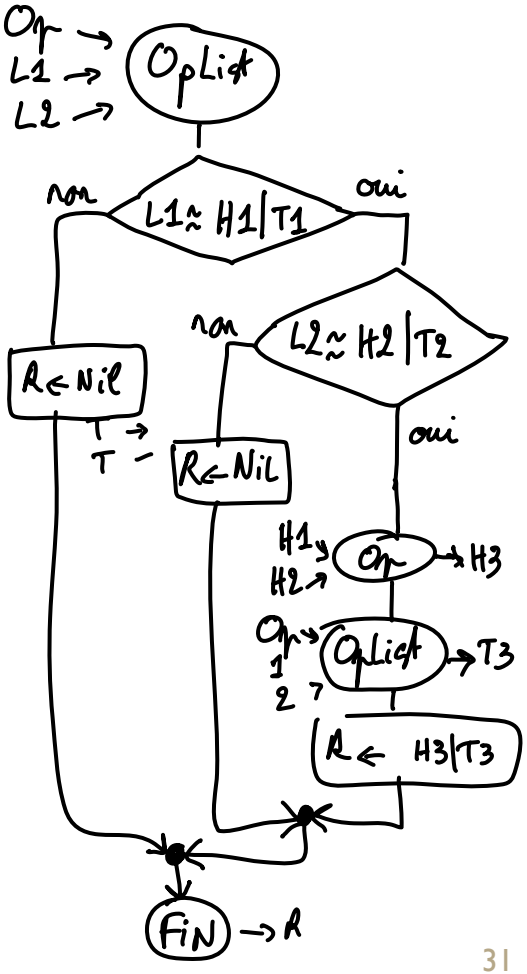
```
fun {OpList Op L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         {Op H1 H2}|{OpList Op T1 T2} end
   else nil end
end
```



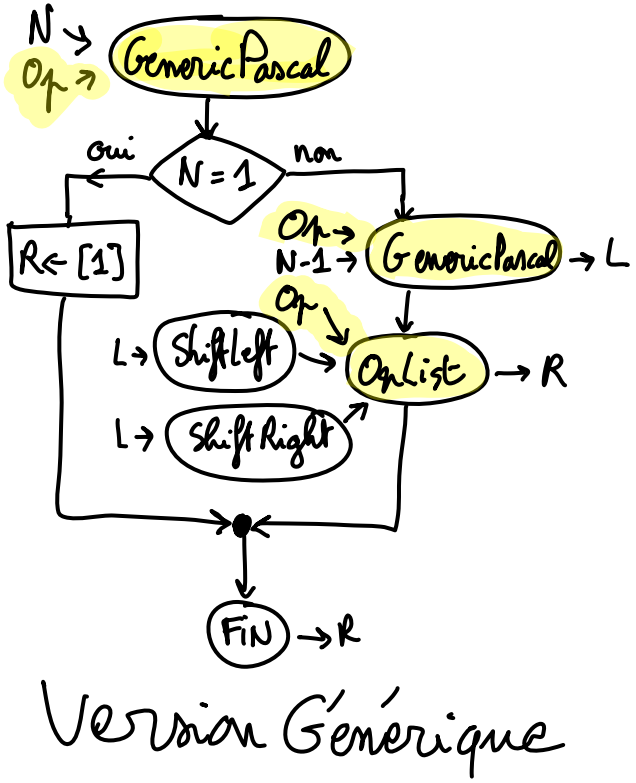
Version optimisée



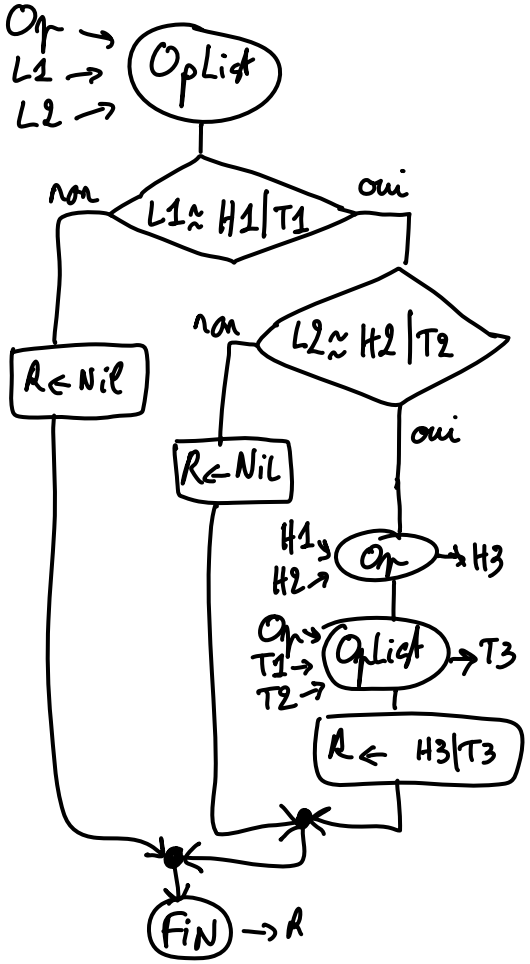
Version Générique



Triangle de Pascal, version générique



Version Générique




```
fun {Add X Y} X+Y end
fun {FastPascal N} {GenericPascal Add N} end
fun {Xor X Y} if X==Y then 0 else 1 end end
```

Programmation concurrente

- Concurrence
  - Plusieurs activités s'exécutant « en même temps » dans un programme ➔ notion de fil (ou thread)
  - Cf architectures multi processeurs et multicores
- Dataflow
  - Si une variable n'est pas liée alors qu'elle est nécessaire pour un calcul
    ➔ suspension du programme

L'état explicite

- Ajouter une sorte de mémoire interne aux fonctions
- Ex. savoir combien de fois la fonction Pascal est appelée
- Cellule mémoire
  - Variable à affectation multiple
  - Opérations
    - Création (allocation de mémoire)
    - Affectation (et plus liaison) ou « écriture »
    - Accès (au contenu) ou « lecture »

Les objets et classes

- Un objet :
  une fonction avec une mémoire interne
  (ou une mémoire avec une ou plusieurs fonctions…)
- Ex.

```
declare
local C in
   C={NewCell 0}
   fun {Bump} C:=@C+1 @C end
   fun {Read} @C end
end
```

Déclaration d'1 objet

```
declare
fun {NewCounter}
C Bump Read in
   C={NewCell 0}
   fun {Bump} C:=@C+1 @C end
   fun {Read} @C end
   counter(bump:Bump read:Read)
end
```

Déclaration d'une classe NewCounter : définie comme une fonction qui contient l'enregistrement de 2 fonctions
⇒ prog. d'ordre supérieur
car les fonctions sont traitées comme valeurs

Création de 2 objets de classe NewCounter
Appel à la méthode Bump du 1er objet.

```
declare
Ctr1={NewCounter}
Ctr2={NewCounter}
{Browse {Ctr1.bump}}
```

La programmation orientée objet

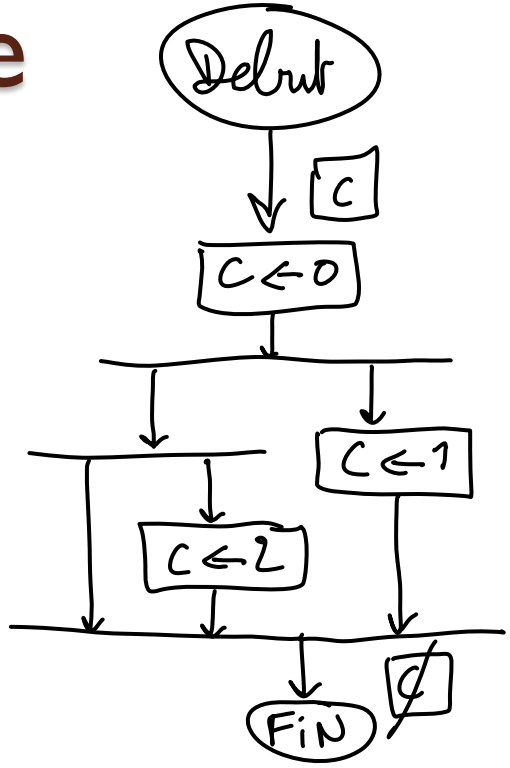
- A base de classes et d'objets (donc notion d'état explicite)
- Respecte aussi
  - L'encapsulation
    - Les variables sont invisibles à l'extérieur de l'objet
  - L'abstraction de données
    - Séparation de l'interface et de l'implémentation
    - L'utilisateur ne connaît que l'interface et pas l'implémentation
  - Le polymorphisme
    - On peut changer l'implémentation, si l'interface reste identique les programmes utilisant la classe continuent de marcher.
  - L'héritage
    - Définition de nouvelles classes à partir d'anciennes, en spécifiant en quoi les nouvelles diffèrent des anciennes

Non-déterminisme

- Utilisation conjointe de
  - La concurrence
  - Et l'état explicite
- Cause : indépendance des fils
  - L'ordre dans lesquels les fils utilisent l'état peut changer d'une exécution à l'autre
  - (Dépendance envers le SE)
- Pb : si le non-déterminisme est observable
  - Il « se voit ».
  - Il a un impact dans le programme (et pas seulement une suspension de programme).

Non-déterminisme observable

```
declare
C={NewCell 0}
thread C:=1 end
thread C:=2 end
```



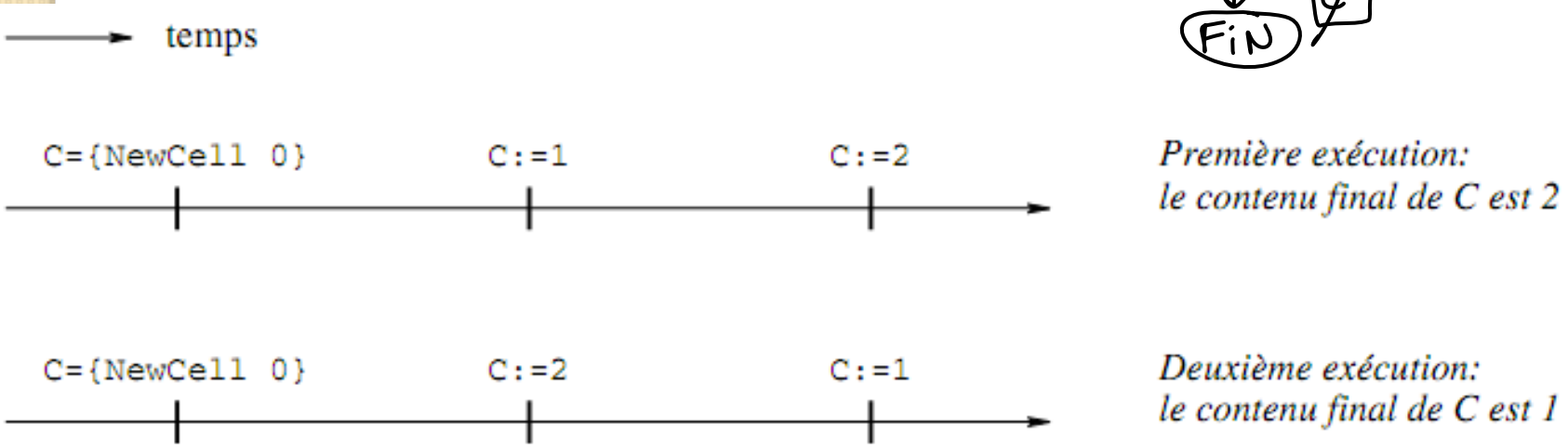
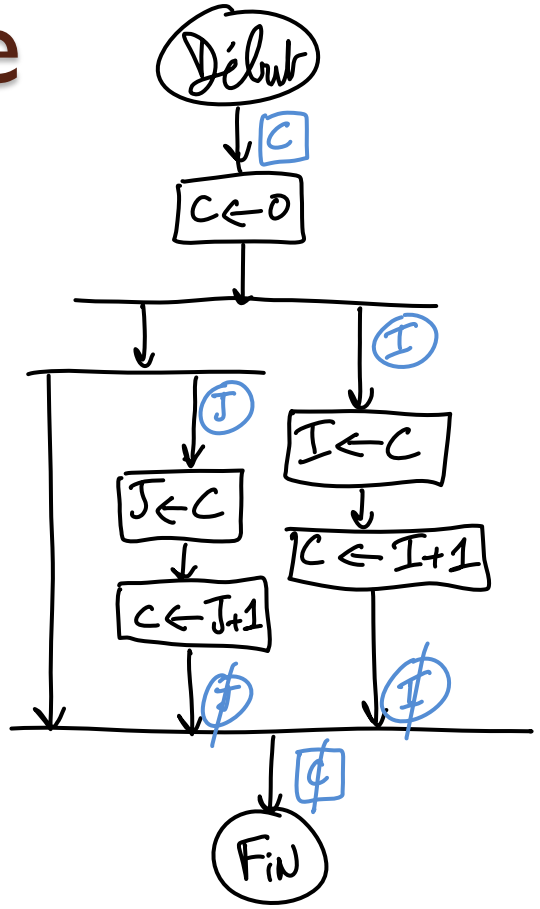


Figure 1.4 Toutes les exécutions possibles du premier exemple non-déterministe.

Non-déterminisme observable

```
declare
C={NewCell 0}
thread I in I=@C C:=I+1 end
thread J in J=@C C:=J+1 end
```



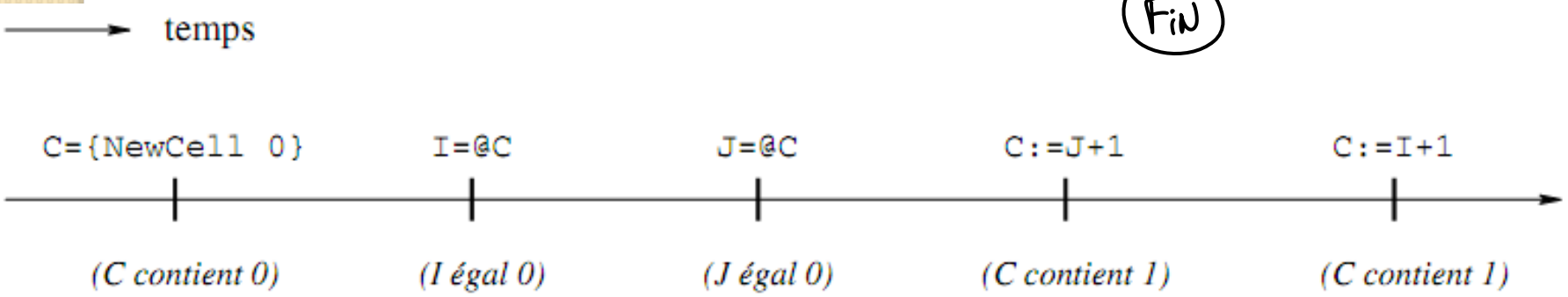


Figure 1.5 Une exécution possible du deuxième exemple non-déterministe.