Macros et procédures “en ligne” (1)

- Les macros ont d'autres différences sémantiques avec les procédures

  1. Catégorie syntaxique de l'expansion possiblement différente de l'appel (idem pour le paramètre formel)

```
#define abs(x) (x<0) ? -x : x
#define echange(x,y) { int t = y; y = x; x = t; }

b = abs(a+1)*2; /* b = (a+1<0) ? -a+1 : a+1*2;     */
if (a < b)      /* if (a < b)                      */
  echange(a,b); /*   { int t = b; b = a; a = t; }; */
else ...        /* else ...  <-- erreur de syntaxe */

#define abs(x) (((x)<0) ? -(x) : (x))
#define echange(x,y) do { int t = y; y = x; x = t; } while (0)

b = (((a+1)<0) ? -(a+1) : (a+1))*2;
if (a < b)
  do { int t = b; b = a; a = t; } while (0);
else ...
```

  2. Évaluation multiple du paramètre actuel (ceci est un problème difficile à contourner en même temps que les conflits de noms)

```
b = abs(f(a));

  /* b = (((f(a)+1)<0) ? -(f(a)+1) : (f(a)+1))*2; */

echange(x[i++],x[j++]);

  /* do {int t=x[j++];x[j++]=x[i++];x[i++]=t;} while (0); */

#define echange(x,y) \
 do { int *px=&(x), *py=&(y), t=*py; *py=*px; *px=t; } while (0)
```


Macros et procédures “en ligne” (2)

- Certains langages permettent d'indiquer qu'une procédure est “**en ligne**”

- Exemple en C++

```
inline void echange (int *x, int *y)
{
  int t = *y;
  *y = *x;
  *x = t;
}
```

- C'est une indication qu'il est **souhaitable** que le corps de cette procédure soit expansé à la place de chaque appel

- Le compilateur est libre d'ignorer cette indication car le **résultat du programme ne changera pas** (i.e. même sens qu'appel de procédure usuel)

Macros et procédures “en ligne” (3)

- Le compilateur doit **renommer les paramètres formels et variables locales** de la procédure dans chaque expansion pour éviter les conflits de noms

```
int t, x;

echange(&t,&x);  /* { int *v601 = &t;
                    int *v602 = &x;
                    int v603 = *v602;
                    *v602 = *v601;
                    *v601 = v603;
                  }
                */
```


Paramètres optionnels

- Ada et C++ permettent de donner une valeur de défaut aux paramètres

- Si le paramètre actuel n'est pas fourni, la valeur de défaut est utilisée

- Exemple en C++, affichage d'un entier dans une certaine base sur un certain fichier de sortie

```
void print_int (int n, int base = 10, FILE *f = stdout)
{
  ...
}

void proc ()
{
  print_int (13);            // decimal sur stdout

  print_int (13, 2);         // binaire sur stdout

  FILE *fich = fopen ("xxx", "w");
  print_int (13, 8, fich);   // octal sur le fichier
  fclose (fich);
}
```

Paramètres nommés

- Ada et Lisp permettent de spécifier les paramètres actuels en les nommant (dans n'importe quel ordre)
- Utile lorsque combiné avec paramètres optionnels et aussi pour documenter le programme
- Exemple en Ada

```
procedure header (page  : in Integer;
                  title : in String  := "";
                  center: in Boolean := True) is
  begin
    ...
  end header;

procedure example is
  begin

    header (100, "", False);

    header (100, center => False);

    header (center => False, page => 100);

  end example;
```


Surdéfinition de procédure

- Ada, C++ et Prolog permettent d'utiliser un même nom pour déclarer plusieurs procédures (**procédures surdéfinies**)
- Le compilateur décide quelle procédure appeler en fonction du **nombre et type des paramètres**
- La déclaration qui **concorde "le mieux"** avec l'appel est choisie
- Exemple en C++

```
void print (int n)  { ... } // V.1
void print (char c) { ... } // V.2

void test ()
{
  print (123);  // V.1
  print ('X');  // V.2
  print (true); // V.1
  print (33.5); // erreur car ambiguë
}
```


Sélection de procédure surdéfinie en C++ (1)

- Cas à 1 paramètre: sélection par ordre de spécificité
  1. Recherche d'une procédure avec paramètre formel de **même type** que paramètre actuel
  2. Nouvelle recherche après **promotion numérique** du paramètre actuel
     - `enum,bool,char,short,int -> int`
     - `long -> long`
     - `float,double -> double`
  3. Nouvelle recherche en considérant les **conversions standard** du paramètre actuel:
     - élargissement, p.e. `char -> long`
     - rétrécissement, p.e. `float -> int`
     - `T* -> void*`

Cas à plus d'un paramètre

- Éliminer les procédures qui sont moins spécifiques qu'une des autres sur au moins un des paramètres (`même` > `prom` > `conv`)
- L'appel est **ambiguë** s'il ne reste pas exactement une procédure
- Exemple avec 2 paramètres

```
void f (char x, int y) { ... } // V.1
void f (int x, char y) { ... } // V.2

void test ()
{             //              V.1         V.2
 f ('A',123); // V.1       même+même   prom+conv
 f (123,'A'); // V.2       conv+prom   même+même
 f (8.7,true);// V.1       conv+prom   conv+conv
 f (123,123); // ambiguë   conv+même   même+conv
}
```

Surdéfinition des opérateurs

- C++ permet de surdéfinir +, -, <<, etc
- Utile pour mieux intégrer les types usagers au langage (nombres, impression)
- Se fait comme une définition de fonction avec le nom "`operator`<*opérateur*>" et au moins un paramètre doit être un `struct` ou `class`
- Exemple: implantation des nombres complexes

```
typedef struct cpx { double reel, imag; } cpx;

cpx operator+ (cpx a, cpx b)
{ cpx r;
  r.reel = a.reel + b.reel;
  r.imag = a.imag + b.imag;
  return r;
}

cpx operator+ (cpx a, double b)
{ cpx t;
  t.reel = b;
  t.imag = 0;
  return a + t;
}

void test ()
{ cpx x, y, z;
  x = (y + (1 + 2)) + z;
}
```


Activation de procédure (1)

- Objectif: expliquer les mécanismes d'activation de procédure et d'accès aux variables
- Problème fondamental: comment associer une valeur à une variable?

  Nom de variable ---> valeur de la variable

- Décomposé en 3 étapes
  1. Nom de variable ---> déclaration de variable ou paramètre
     - **règles de portée**
  2. Déclaration de variable ---> emplacement mémoire
     - **activation courante**
  3. Emplacement mémoire ---> valeur
     - **état de la mémoire**

Activation de procédure (2)

- La deuxième étape est nécessaire pour traiter les procédures **récursives**
- Déf: une procédure est **récursive** s'il est possible que pendant une de ses activation elle soit activée à nouveau
  - **Directement** récursive: `f ---> f`
  - **Indirectement** récursive: `f ---> g ---> f`
- Exemple en Pascal

```
program recursif;

var x : integer;

function f(n:integer)
   : integer;
 begin
  if n <= 1 then
   f := 1
  else
   f := n * f(n-1);
 end;

begin
x := f(3);
writeln(x);
end.
```


Activation de procédure (3)

- Il y a 3 activations de `f` et **chaque activation a sa propre instance de la variable** `n`

```
program recursif;

      +-----+
    x |  ?  |
      +-----+

var x : integer;

           +-----+                 +-----+                 +-----+
         n |  3  |               n |  2  |               n |  1  |
           +-----+                 +-----+                 +-----+

function f(n:integer)    function f(n:integer)    function f(n:integer)
   : integer;               : integer;               : integer;
 begin                    begin                    begin
  if n <= 1 then           if n <= 1 then           if n <= 1 then
   f := 1                   f := 1                   f := 1
  else                     else                     else
   f := n * f(n-1);         f := n * f(n-1);         f := n * f(n-1);
 end;                     end;                     end;

begin
x := f(3);
writeln(x);
end.
```

- La récursivité implique l'allocation **dynamique** des variables locales et paramètres formels (en général le nombre d'appels récursifs n'est pas connu statiquement)

Activation: cas non-récursif (1)

- L'espace pour stocker les paramètres et variables locales des procédures non-récursives peut être **alloué statiquement** (p.e. FORTRAN)

- Exemple, compilation de Pascal à C (qui est utilisé comme un "langage machine" de haut niveau)

```
  Programme Pascal                C sans paramètres/résultat

program non_recursif;         |  int x;
                              |
var x : integer;              |  int mult_n, mult_m, mult_res;
                              |
function mult(n,m:integer)    |  void mult()
   : integer;                 |  {
 begin                        |    mult_res = mult_n * mult_m;
  mult := n*m;                |  }
 end;                         |
                              |  void non_recursif()
begin                         |  {
x := mult(2,3);               |    mult_n = 2;
writeln(x);                   |    mult_m = 3;
x := mult(4,5);               |    mult();
writeln(x);                   |    x = mult_res;
end.                          |    printf("%d",x);
                              |
                              |    mult_n = 4;
                              |    mult_m = 5;
                              |    mult();
                              |    x = mult_res;
                              |    printf("%d",x);
                              |  }
```


Activation: cas non-récursif (2)

- Pour expliquer plus en détail les **transferts de contrôle** requis pour une activation, nous utilisons les "`goto` **calculés**" qui existent sur certains compilateurs C (`gcc`)
  - `&&`<*étiqu*> => pointeur (de type `void*`) vers l'étiquette
  - `goto *`<*ptr_étiqu*>`;` => transfert de contrôle à l'étiquette
  - Exemple

```
void *dest;

if (x<0) dest = &&neg; else dest = &&pos;

goto *dest;

neg: ...

pos: ...
```


Activation: cas non-récursif (3)

- La procédure appelante doit indiquer à la procédure appelée **où il faut retourner le contrôle après l'activation** (**adresse de retour**)

- L'adresse de retour est simplement un **paramètre implicite** de toutes les procédures

```
  Programme Pascal                C sans procédures

program non_recursif;         |  int x;
                              |
var x : integer;              |  void *mult_ret;
                              |  int mult_n, mult_m, mult_res;
function mult(n,m:integer)    |
   : integer;                 |  void non_recursif()
 begin                        |  {
  mult := n*m;                |    mult_n = 2;
 end;                         |    mult_m = 3;
                              |    mult_ret = &&point1;
begin                         |    goto mult;
x := mult(2,3);               |  point1:
writeln(x);                   |    x = mult_res;
x := mult(4,5);               |    printf("%d",x);
writeln(x);                   |
end.                          |    mult_n = 4;
                              |    mult_m = 5;
                              |    mult_ret = &&point2;
                              |    goto mult;
                              |  point2:
                              |    x = mult_res;
                              |    printf("%d",x);
                              |
                              |    return;
                              |
                              |  mult:
                              |    mult_res = mult_n * mult_m;
                              |    goto *mult_ret;
                              |  }
```


Activation: cas récursif (1)

- Ceci ne fonctionne pas avec les procédures récursives

```
program recursif;         |  int x;
                          |
var x : integer;          |  void *f_ret;
                          |  int f_n, f_res;
function f(n:integer)     |
   : integer;             |  void recursif()
 begin                    |  {
  if n <= 1 then          |    f_n = 3;
   f := 1                 |    f_ret = &&point1;
  else                    |    goto f;
   f := n * f(n-1);       |  point1:
 end;                     |    x = f_res;
                          |    printf("%d",x);
begin                     |    return;
x := f(3);                |
writeln(x);               |  f:
end.                      |    if (f_n <= 1)
                          |      f_res = 1;
                          |    else
                          |      { f_n = f_n-1;
                          |        f_ret = &&point2;
                          |        goto f;
                          |      point2:
                          |        f_res = f_n * f_res;
                          |      }
                          |    goto *f_ret;
                          |  }
```

- La valeur de `f_n` et de `f_ret` au retour de `f` sont incorrectes

Activation: cas récursif (2)

- L'état d'une activation est conservé dans un **bloc d'activation** qui stocke
  - Les paramètres formels (incluant l'adresse de retour)
  - Les variables locales et temporaires (introduites par le compilateur)
  - Le résultat de la fonction (pas vraiment nécessaire car allocation statique possible)
  - Le **lien dynamique** qui chaîne les blocs des activations qui n'ont pas encore terminées
- Le lien dynamique permet de retrouver le bloc de la procédure appelante au moment du retour de la procédure courante

Activation: cas récursif (3)

```
program recursif;

var x : integer;

function f(n:integer)
   : integer;
 begin
  if n <= 1 then
   f := 1
  else
   f := n * f(n-1);
 end;

begin
x := f(3);
writeln(x);
end.
```

```
typedef struct bloc
  { int n;
    void *ret;
    struct bloc *dyn;
  } bloc;

bloc *bac, *p;
int x;
int f_res;

void recursif()
{ bac = NULL;

  p = (bloc*)malloc(sizeof(bloc));
  p->n = 3;
  p->ret = &&point1;
  p->dyn = bac;
  bac = p;
  goto f;
point1:
  p = bac->dyn;
  free(bac);
  bac = p;
  x = f_res;
  printf("%d",x);
  return;

f:
  if (bac->n <= 1)
    f_res = 1;
  else
    { p = (bloc*)malloc(sizeof(bloc));
      p->n = bac->n-1;
      p->ret = &&point2;
      p->dyn = bac;
      bac = p;
      goto f;
    point2:
      p = bac->dyn;
      free(bac);
      bac = p;
      f_res = bac->n * f_res;
    }
  goto *bac->ret;
}
```


Activation: gestion mémoire (1)

- Quand faut-il **récupérer** l'espace occupé par le bloc d'activation?
- La plupart des langages le font **au retour de l'activation** (exceptions: Modula-3, Oberon, Lisp, Scheme)
- Un problème relié à ce choix, en C, c'est que l'opérateur "&" peut engendrer des **pointeurs fous**

```
int *p;

void print () { printf ("%d\n", *p); }

void proc (int n) { p = &n; print (); }

void main () { proc (123); print (); }
```

- Dans ce cas, il est possible d'allouer les blocs d'activation **sur une pile** car le bloc d'activation de l'appelée est seulement actif pendant l'activation de l'appelant
- Le **lien dynamique** n'a plus besoin d'être stocké explicitement

Activation: gestion mémoire (2)

```
program recursif;

var x : integer;

function f(n:integer)
   : integer;
 begin
  if n <= 1 then
   f := 1
  else
   f := n * f(n-1);
 end;

begin
x := f(3);
writeln(x);
end.
```

```
typedef struct bloc
  { int n;
    void *ret;
  } bloc;

bloc pile[100], *bac;
int x;
int f_res;

void recursif()
{
  bac = pile;

  (bac+1)->n = 3;
  (bac+1)->ret = &&point1;
  bac++;
  goto f;
point1:
  bac--;
  x = f_res;
  printf("%d",x);
  return;

f:
  if (bac->n <= 1)
    f_res = 1;
  else
    { (bac+1)->n = bac->n-1;
      (bac+1)->ret = &&point2;
      bac++;
      goto f;
    point2:
      bac--;
      f_res = bac->n * f_res;
    }
  goto *bac->ret;
}
```

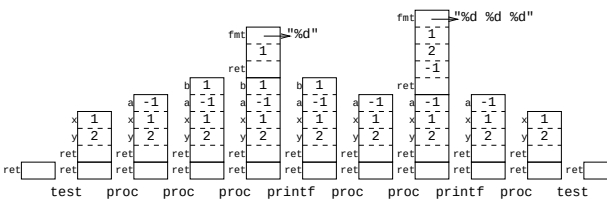
Activation: gestion mémoire (3)

- L'allocation et la récupération des blocs d'activation peuvent se faire **d'un seul coup** (tel que fait précédemment) ou **incrémentallement**
- L'allocation et récupération incrémentale se fait en partie dans l'appelant et en partie dans l'appelé
- Par exemple, les compilateurs C font normalement
  - L'allocation d'un bloc contenant les paramètres **dans l'appelant** (empilés dans l'ordre droite à gauche)
  - L'**extension** de ce bloc avec les variables locales **dans l'appelé**
  - La **contraction** de ce bloc en quittant la portée des variables locales **dans l'appelé**
  - La récupération du bloc au retour de l'activation **dans l'appelant**

Activation: gestion mémoire (4)

- Exemple en C

```
void proc (int x, int y)
{ int a;
  a = x-y;
  if (a < 0)
    {
      int b;
      b = a*a;
      printf ("%d", b); /* 1 */
    }
  printf ("%d %d %d", x, y, a); /* 1 2 -1 */
}

void test () { proc (1, 2); }
```

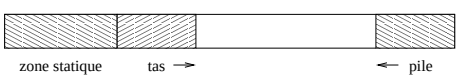


- Empiler les paramètres de droite à gauche permet une implantation simple des fonctions à **arité variable** comme `printf`:

```
int printf (char* fmt, ...)  /* un véritable "..." */
{/* utiliser "fmt" pour obtenir les autres paramètres */}
```


Activation: procédures imbriquées (1)

- Une approche répandue pour gérer la mémoire est de la subdiviser en 3 zones
  1. **zone statique**: variables globales et code
  2. **pile**: paramètres et variables locales
  3. **tas**: autres allocations dynamiques

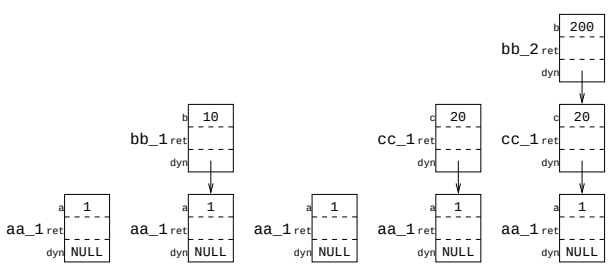


- L'accès à une **variable globale** est simple car l'adresse de la variable dans la zone statique est connue à la compilation
- L'accès à une **variable non-globale** demande de **trouver le bloc d'activation** la contenant et d'accéder au champ approprié de ce bloc
  - En C et FORTRAN ceci est relativement simple puisque la variable est nécessairement **dans le bloc d'activation courant**
  - Avec des procédures imbriquées ceci n'est plus vrai

Activation: procédures imbriquées (2)

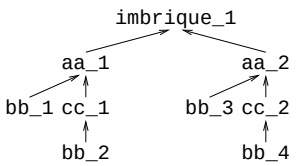
- Exemple en Pascal avec lien dynamique

```
program imbrique;

  var x : integer;

  procedure aa (a : integer);

    procedure bb (b : integer);
      begin
        writeln (b + a);
      end;

    procedure cc (c : integer);
      begin
        bb (c*10);
      end;

  begin  bb (10);  cc (20);  end;

begin  aa (1); (* 11 201 *)  aa (2); (* 12 202 *) end.
```

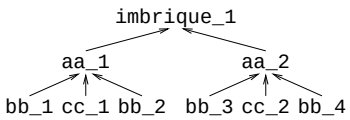


Activation: procédures imbriquées (3)

- Déf: le **parent dynamique** d'une activation X c'est l'activation Y qui a créée l'activation X



- Déf: le **parent lexical** d'une activation X c'est l'activation Y qui a déclarée la procédure dont l'appel a créé l'activation X

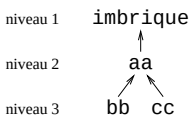


- Le **lien dynamique** chaîne les parents dynamiques et le **lien statique** chaîne les parents statiques

Activation: procédures imbriquées (4)

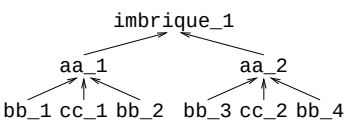
- La chaîne statique est utilisée pour trouver rapidement le bloc d'activation contenant une variable à accéder

- Déf: le **niveau d'imbrication** d'une procédure c'est sa profondeur dans l'arbre d'imbrication (avec racine = 1)



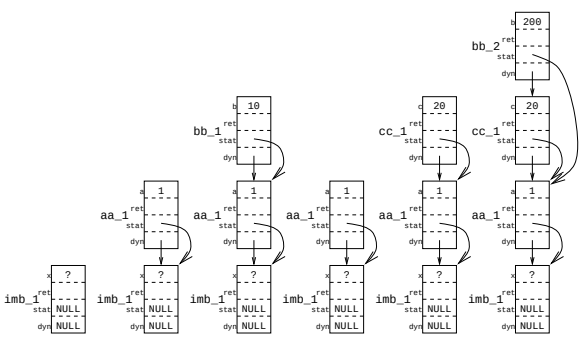
- Si l'exécution se trouve dans la procédure X de niveau x et qu'il faut accéder à une variable V déclarée dans la procédure englobante Y de niveau y alors

  – le bloc contenant V est à une distance de $x - y$ dans la chaîne statique par rapport au bloc d'activation courant



Activation: procédures imbriquées (5)

- Blocs d'activation avec liens statiques et dynamiques (si on traite le programme principal comme une procédure qui englobe tout)



- Accès à la variable `a` dans la procédure `bb`?

  – `a` est déclarée dans `aa` (niveau 2)

  – `bb` est au niveau 3

  – donc il faut sauter $3-2=1$ liens statiques pour obtenir le bloc qui contient `a`

```
printf ("%d\n", bac->b + bac->stat->a); /* writeln (b + a);
```


Activation: procédures imbriquées (6)

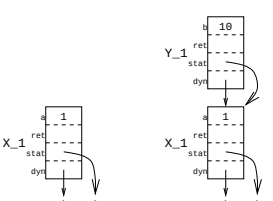
- Comment construire la **chaîne statique**?

- Si la procédure X active la procédure Y il faut créer un bloc d'activation et initialiser le lien statique à un pointeur vers le **parent statique** de l'activation

- Comment trouver le **parent statique**?

- Cas 1: parent statique = **bloc appelant**

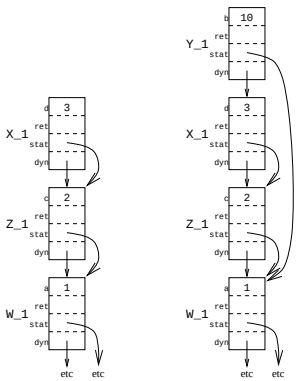
```
procedure X (a : integer);

  procedure Y (b : integer); (* Y est sous-procédure de X *)
    begin
      writeln (b + a);
    end;

  begin  Y (10);  end;
```



Activation: procédures imbriquées (7)

- Cas 2: parent statique est **dans la chaîne statique** (a un niveau d'imbrication connu à la compilation)

```
procedure W (a : integer);

  procedure Y (b : integer); (* Y est sous-procédure d'une *)
    begin                    (* procédure englobante de X  *)
      writeln (b + a);
    end;

  procedure Z (c : integer);

    procedure X (d : integer);
      begin  Y (10);  end;

    begin  X (3);  end;

  begin  Z (2);  end;
```



Activation: procédures imbriquées (8)

- Il faut remonter la chaîne statique d'une distance de $x - y + 1$ pour trouver le parent statique (où x et y sont les niveaux d'imbrication de X et Y)

- Dans l'exemple $x - y + 1 = 2$

```
p = bac;
bac = ...; /* allocation du nouveau bloc */
bac->dyn  = p;
bac->stat = p->stat->stat;
```

- Note: le cas 1 est un cas spécial du cas 2

- Dans l'exemple du cas 1, $x - y + 1 = 0$

```
p = bac;
bac = ...; /* allocation du nouveau bloc */
bac->dyn  = p;
bac->stat = p;
```


Activation: procédures imbriquées (9)

- Cas 3: procédure passée en paramètre (possible en C, Pascal, SIMULA, Scheme)

```
procedure fn_en_param;

  function derive (function f(w:real):real, x:real) : real;
    begin
      derive := (f(x+0.0001) - f(x)) / 0.0001;
    end;

  function carre (y : real) : real;
    begin  carre := y*y;  end;

  procedure test (n : integer);

    function puiss_n (z : real) : real;
        var temp : real; i : integer;
      begin
        temp := 1;
        for i := 1 to n do temp := temp*z;
        puiss_n := temp;
      end;

  begin  writeln (derive (puiss_n, 10));  end;

begin
writeln (derive (carre, 10)); (* 20.0001 *)
test (3);                     (* 300.003 *)
test (4);                     (* 4000.06 *)
end;
```

- Problème 1: le niveau d'imbrication de l'appelé pour un appel donné n'est **pas constant**

- Problème 2: le parent statique de l'appelé n'est **pas nécessairement dans la chaîne statique**

Activation: procédures imbriquées (10)

- Un paramètre procédural est représenté par un enregistrement contenant:

  1. pointeur vers le code exécutable (`code`)
  2. pointeur vers le parent statique (`parent`)

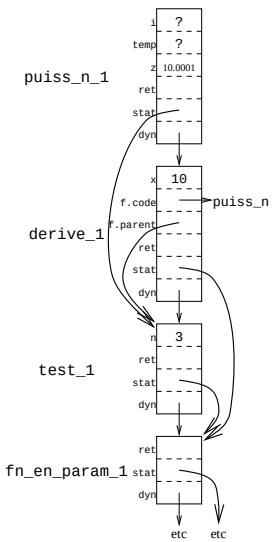
- Protocole d'activation (en supposant un appel au paramètre procédural `f`)

```
p = bac;
bac = ...; /* allocation du nouveau bloc */
bac->dyn  = p;
bac->stat = p->f.parent;
goto *p->f.code;
```

- C'est lorsqu'on passe un paramètre procédural que le parent statique est cherché (avec le même algorithme que le cas 2)

Activation: procédures imbriquées (11)

- Blocs d'activation créés

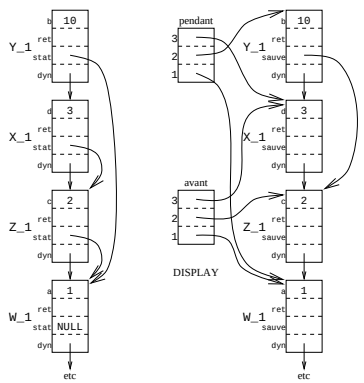


Activation: procédures imbriquées (12)

- Il existe une autre approche pour gérer la chaîne statique
- Déf: un **"display"** c'est une table de pointeurs vers des blocs d'activation
- Plutôt que de représenter la chaîne statique comme une liste de blocs, utiliser le display:
  - `display[`*niveau*`]` = pointeur vers bloc dans la chaîne à ce niveau d'imbrication
- L'accès à un bloc se fait en temps constant quel que soit le niveau d'imbrication du bloc
- L'accès à une variable `V` dans un bloc au niveau `n`: `display[n]->V`
- Taille du display = niveau d'imbrication maximal

Activation: procédures imbriquées (13)

- Comparaison avec liste de blocs



- Activation de Y par X (cas statique)

```
p = bac;
bac = ...; /* allocation du nouveau bloc */
bac->ret = &&retour;
bac->dyn = p;
goto Y;

Y: bac->sauve = display[y]; /* sauver pointeur */
   display[y] = bac;
   ...
   display[y] = bac->sauve; /* ancien état */
   goto *bac->ret;
```


Activation: procédures imbriquées (14)

- Le cas des procédures passées en paramètre est plus complexe car il y a possiblement **plus qu'une entrée** du display qui doit changer
- Un paramètre procédural est représenté par un enregistrement contenant:
  1. pointeur vers le code exécutable (`code`)
  2. display représentant la chaîne statique du parent statique (`parent`)
- Activation de Y par X (cas paramètre procédural)

```
bac->sauve_display[*] = display[*]; /* sauver display */
p = bac;
bac = ...; /* allocation du nouveau bloc */
bac->ret = &&retour;
bac->dyn = p;
display[*] = Y.parent[*]; /* installer chaîne du parent */
goto *Y.code;

retour:
bac = bac->dyn;
display[*] = bac->sauve_display[*]; /* ancien état */
```