Compilation de la sélection par cas

- Techniques adaptées à diverses situations:

1. **Peu de cas ou prépondérance d'un petit nombre de cas**: utilisation d'une cascade de `ifs` ordonnés par probabilité d'occurence

```
case x of                              if x=2 then y:=2
  1: y:=1; (* P(x=1) = .1 *)           else if x=3 then y:=3
  2: y:=2; (* P(x=2) = .5 *)   =>      else if x=1 then y:=1
  3: y:=3; (* P(x=3) = .3 *)           else y:=4
  4: y:=4; (* P(x=4) = .1 *)
end                                    (* #ifs = 1*.5 + 2*.3 + 3*.2 *)
```

2. **Intervalle des cas est dense**: utilisation d'un tableau d'adresses

```
case x of         goto_calculé T[x-2]
  2: y:=2;        e2: y:=2; goto fin;
  4: y:=4;   =>   e4: y:=4; goto fin;
  5: y:=5;        e5: y:=5; goto fin;
  9: y:=9;        e9: y:=9; goto fin;
end               fin:
                  (* T[0]=e2 T[1]=err T[2]=e4 ... T[7]=e9 *)
```

3. **Intervalle des cas est épars**: utilisation d'une table d'adressage dispersé parfaite (perfect hashing), i.e. sans collisions

```
case x of         goto_calculé T[H(x)]
 10: y:=10;       e10: y:=10; goto fin;
 48: y:=48;  =>   e48: y:=48; goto fin;
 52: y:=52;       e52: y:=52; goto fin;
 99: y:=99;       e99: y:=99; goto fin;
end               fin:
                  (* T[H(10)]=e10 T[H(48)]=e48 ... *)
                  (* avec H(x) = x mod 5 *)
```


Procédures

- Déf: **procédure** = fragment de programme paramétré
- Déf: **signature** d'une procédure = type des paramètres (et du résultat dans le cas d'une fonction)
- Déf: **corps** d'une procédure = opérations associées à la procédure
- Une définition de procédure donne généralement

1. Le nom de la procédure
2. Sa signature
3. Le nom des paramètres formels
4. Le corps

- Exemple en C

```
double demi (int n)  /* signature: int -> double */
{ return n*0.5; }
```


Appel et activation de procédure

- Déf: **appel de procédure** = utilisation de la procédure dans le texte du programme
- Déf: **activation de procédure** = exécution du corps de la procédure
- Exemple en C: 1 appel et 5 activations de `demi`

```
void test ()
{ int i;
  double x = 0;
  for (i=1; i<=5; i++) x += demi (i);
}
```

- 2 procédures impliquées par activation:

1. La procédure **appelante** contient l'appel
2. La procédure **appelée** est celle qui est activée par l'appel

- Un appel de procédure contient les **paramètres actuels**
- Les **paramètres formels** sont les représentants des paramètres actuels dans le corps de l'appelée

Bénéfices des procédures

1. **Abstraction**: permet d'exprimer des opérations abstraites, i.e. plus proches des besoins de l'application (trier, chercher, ...)
2. **Isoler l'implantation**: si la spécification de la procédure est bien conçue (c'est-à-dire assez abstraite), le corps de la procédure peut être changé et amélioré sans avoir à changer le reste du programme
3. **Modularité**: un programme peut être découpé en morceaux qui sont plus faciles à comprendre indépendamment
4. **Librairies**: groupe de procédures d'usage général réutilisable par divers programmes (`stdio`, `Xlib`, ...)

Mode de passage de paramètre

- Déf: **mode de passage de paramètre** = lien qui existe entre le paramètre formel et le paramètre actuel

- Normallement le choix du mode de passage peut se fait indépendamment pour chaque paramètre

- Les 4 modes les plus répandus

  1. **par valeur** (existe dans presque tous les langages)
  2. **par référence** (Pascal, Modula-2, C++)
  3. **par valeur-résultat** (Ada)
  4. **par nom** (Algol, SIMULA)

Passage par valeur

- Le paramètre actuel est une **expression "R-value"** (qui peut se trouver du côté droit d'une affectation)

- Le paramètre formel est une **variable** créée pour l'activation et initialisée avec la valeur du paramètre actuel

- Exemple en C

```
int carre (int x)  /* x passé par valeur */
{ x = x*x; return x; }

void test ()
{ int n = 3;
  printf ("%d", carre (n));   /* 9 */
  printf ("%d", n);           /* 3 */
  printf ("%d", carre (n+1)); /* 16 */
}
```

- Donc, les affectations au paramètre formel **n'affectent pas** le paramètre actuel

Passage par référence

- Le paramètre actuel est une **variable** ou tout autre emplacement mémoire (expression "**L-value**" pouvant se trouver du côté gauche d'une affectation)

- Le paramètre formel désigne le **même emplacement mémoire** que celui désigné par le paramètre actuel au moment de l'activation (i.e. c'est un **alias** de l'emplacement mémoire)

- Exemple en C++

```
int carre (int &x)  /* x passé par référence */
{ x = x*x; return x; }

void test ()
{ int n = 3;
  printf ("%d", carre (n)); /* 9 */
  printf ("%d", n);         /* 9 */
}
```

- Le passage par valeur d'un pointeur est équivalent

```
int carre (int *x)  /* x est un pointeur passé par valeur */
{ *x = (*x)*(*x); return *x; }

void test ()
{ int n = 3;
  printf ("%d", carre (&n)); /* 9 */
  printf ("%d", n);          /* 9 */
}
```


Passage par valeur-résultat

- Le paramètre actuel est une **variable** ou tout autre emplacement mémoire (**L-value**)

- Le paramètre formel est une **variable** créée pour l'activation et initialisée avec la valeur du paramètre actuel

- Au retour de la procédure, le paramètre formel est copié dans le paramètre actuel

- Exemple en Ada

```
i, n : integer;

procedure add2 (x : in out integer) is
  begin
    x := x+i;
    x := x+i;
  end;

n := 0;  i := 1;
add2 (n);
put (n); -- imprime 2 (serait 2 par référence)
add2 (i);
put (i); -- imprime 3 (serait 4 par référence)
```

- Permet accès plus rapide au paramètre formel que passage par référence (évite indirection)

- Sens plus simple pour les systèmes concurrents

Passage par nom (1)

- Le paramètre actuel est une **L-value** ou une **R-value**
- Une utilisation du paramètre formel évalue le paramètre actuel comme L-value (si à gauche d'une affectation) sinon comme R-value
- Le paramètre actuel est évalué **à chaque accès** et dans le contexte de l'appelant
- Exemple en SIMULA

```
integer i, n;
integer array t(1:2);

procedure add2 (x); name x; integer x;
  begin
    x := x+1;
    i := i+1;
    x := x+1;
  end;

n := 0;  i := 1;
add2 (n);
outint (n,0); ! imprime 2 (serait 2 par référence) ;

n := 0;  i := 1;  t(1) := 0;  t(2) := 0;
add2 (t(i));
outint (t(1),0); ! imprime 1 (serait 2 par référence) ;
outint (t(2),0); ! imprime 1 (serait 0 par référence) ;
```


Passage par nom (2)

- Les subtilités de la sémantique du passage par nom peuvent être exploitées pour créer des **nouvelles structures de contrôle**
- Exemple: calcul d'une somme générale, tel que

$$\sum_{i=1}^{10} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{1024}$$

```
real procedure somme (index, bas, haut, valeur);
    name index, valeur;
    integer index, bas, haut;
    real valeur;
  begin
    integer j;
    real s;
    s := 0;
    for j := bas step 1 until haut do
      begin
        index := j;
        s := s + valeur;
      end;
    somme := s;
  end;

integer i;

outfix (somme (i,1,10,1/2**i), 10, 15); ! 0.9990234375 ;
outfix (somme (i,1,10,i), 1, 4);        ! 55.0 ;
```


Noms et portée (1)

- Les noms sont utilisés pour identifier des types, variables, procédures, etc. (**identificateurs**)
- Dans la plupart des langages un même nom peut servir à désigner **plusieurs chôses**
- Les **règles de portée** d'un langage spécifient ce qui est désigné par un nom, en fonction du contexte de son utilisation
- Déf: une **déclaration** introduit un nouveau sens pour un nom
  - déclaration de type, de variable, de paramètre, de procédure, ...
  - Exemple: `int x;` => déclaration de variable qui introduit le nom "`x`"

Noms et portée (2)

- Déf: la **portée** d'une déclaration = la **région** du programme où l'utilisation du nom déclaré désigne cette déclaration
- Exemple en C

```
int carre (int x)
{ return x*x; }

int f (short x)
{ return carre (carre (x/2)); }
```

Portée de "`int x`" = corps de `carre`, portée de "`int carre (int x)`" = corps de `carre` et `f`

- La portée à un certain point du programme correspond à une fonction, p.e. dans le corps de `carre`:

```
utilisation   portée
d'un nom    --------->  déclaration

   x                      int x
  carre                 int carre (int x)
```

- Les deux sortes de portée les plus répandues:

1. **Portée lexicale** (plupart des langages)
2. **Portée dynamique** (Perl, APL, TeX, Lisp)

Portée lexicale (1)

- **Règle:** un nom désigne la déclaration englobante la plus proche **textuellement**

- Donc, une déclaration locale à un bloc (ou procédure) a précéance, à l'intérieur de ce bloc, sur toute déclaration du même nom externe à ce bloc

- Exemple en C

```
char *x = "allo";          /* déclaration 1 de x */

void proc1 (int x, int y)  /* déclaration 2 de x */
{ if (x > y)
    {
      int x = 2*y;         /* déclaration 3 de x */
      printf ("%d\n", x);  /* x --> décl 3 */
    }
  printf ("%d\n", x);      /* x --> décl 2 */
}

void proc2 ()
{ proc1 (3, 2);
  printf ("%s\n", x);      /* x --> décl 1 */
}
```

- Note: en C, une "déclaration" de variable n'inclus pas l'initialisation (i.e. la portée de "`int x`" débute après le = dans "`int x = x*y;`")

Portée lexicale (2)

- Dans certains langages il est permis de déclarer une procédure dans une autre (**procédures imbriquées**), p.e. Pascal, Modula-2, Ada, SIMULA, Scheme, Perl mais pas C et FORTRAN

- La portée des déclarations locales à une procédure s'étend aux sous-procédures

- Exemple en Pascal

```
procedure A;

  var x : char; (* variable locale de A *)

  procedure B;
    begin
      writeln(x);
    end;

  procedure C;

    var x : char; (* variable locale de C *)

    begin  x := 'C';  B;  end;

begin
  x := 'A';
  B; (* imprime A *)
  C; (* imprime A *)
  B; (* imprime A *)
end;
```


Portée dynamique (1)

- **Règle:** un nom désigne la déclaration englobante la plus proche **dans la chaîne d'activations**

- Donc, si la procédure A appèle la procédure B, une déclaration qui est visible dans A le sera également dans B (à moins que B déclare le même nom)

- Exemple en Perl

```
sub A
  { local $x;

    sub B
      { printf ("%s\n", $x);
      }

    sub C
      { local $x = "C";
        B;
      }

    $x = "A";
    B;  # imprime A
    C;  # imprime C
    B;  # imprime A
  }
```

- Note: en Perl, l'utilisation de "`my`" à la place de "`local`" donne la portée lexicale

Portée dynamique (2)

- En général la portée dynamique est à éviter car sa sémantique dépend de l'ordre d'exécution

- Un cas où la portée dynamique est appropriée, c'est pour **passer un paramètre implicite** à toutes les procédures appelées pendant l'activation courante

- Exemple en C: changer la destination d'impression

```
void print_char (char c, FILE* output)
{ fputc (c, output); }

void print_str (char* msg, FILE* output)
{ while (*msg != '\0')
    print_char (*msg++, output);
  print_char ('\n', output);
}

void print_page (char** t, int n, FILE* output)
{ int i;
  for (i=0; i<n; i++)
    print_str (t[i], output);
}

char* p[3] = { "ligne 1", "ligne 2", "ligne 3" };

void reports ()
{ FILE* fich;

  print_page (p, 3, stdout);

  fich = fopen ("rapport", "w");
  print_page (p, 3, fich);
  fclose (fich);
}
```

Portée dynamique (3)

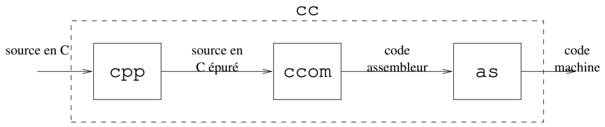
- Si C utilisait la portée dynamique:

```
void print_char (char c)
{ fputc (c, output); }

void print_str (char* msg)
{ while (*msg != '\0')
    print_char (*msg++);
  print_char ('\n');
 }

void print_page (char** t, int n)
{ int i;
  for (i=0; i<n; i++)
    print_str (t[i]);
}

char* p[3] = { "ligne 1", "ligne 2", "ligne 3" };

void reports ()
{ FILE* output;

  output = stdout;
  print_page (p, 3);

  output = fopen ("rapport", "w");
  print_page (p, 3);
  fclose (output);
}
```

- L'équivalent peut se faire avec une variable globale dans un langage non-concurrent
- La portée lexicale se compile mieux (programmes plus rapides) que la portée dynamique car la correspondance nom–déclaration est connue à la compilation

Macros et portée dynamique (1)

- Le compilateur C (cc) est en fait composé de plusieurs programme qui traitent à leur tour le programme à compiler



- cpp est le "préprocesseur C" qui traite toutes les directives de préprocesseur "#" et les macros (le résultat est un programme C sans aucune directives de préprocesseur)
  - `#include "..."`: inclusion de fichier
  - `#if/#ifdef ...`: compilation conditionnelle
  - `#define ...`: définition de macro
- Déf: une **macro** c'est une sorte de procédure dont l'appel sera traité par le préprocesseur, en remplaçant l'appel par le corps de la macro avec une **substitution textuelle** des paramètres

Macros et portée dynamique (2)

- Exemple

```
   Source                         Source après prétraitement

#include <stdio.h>             | int printf (char *format, ...);
                               | ... reste de /usr/include/stdio.h
                               |
#define N 10                   |
#define MIN(x,y) (x<y)?x:y     |
                               |
#if N < 5                      |
                               |
int t[N];                      |
                               |
#else                          |
                               |
int t[N*2];                    | int t[10*2];
                               |
#endif                         |
                               |
int min3(int a,int b,int c)    | int min3(int a,int b,int c)
{                              | {
  a = MIN(a,b);                |   a = (a<b)?a:b;
  a = MIN(a,c);                |   a = (a<c)?a:c;
  return a;                    |   return a;
}                              | }
                               |
int main ()                    | int main ()
{                              | {
#ifdef sun                     |
  printf ("version SUN\n");    |
#endif                         |
                               |
#ifdef linux                   |
  printf ("version linux\n");  |   printf ("version linux\n");
#endif                         |
                               |
  return 0;                    |   return 0;
}                              | }
```


Macros et portée dynamique (3)

- Les macros sont souvent utilisées au lieu de procédures pour **éviter le coût d'activation** (transferts de contrôle, copie des paramètres, ...)
- Le traitement des macros par expansion textuelle peut cependant **accroître la taille de l'exécutable**
- De plus, la substitution textuelle des paramètres cause une sorte de **portée dynamique** lorsque la macro contient des déclarations, ce qui peut produire des résultats inattendus
- Exemple

```
#define echange(x,y) { int t = y; y = x; x = t; }

void proc (int a, int b, int t, int t2)
{ echange(a,b);   /* { int t = b;  b = a;  a = t; }; */
  echange(t,t2);  /* { int t = t2; t2 = t; t = t; }; */

  /* le deuxieme appel ne change pas t et t2 */
}
```