Typage et règles de visibilité

Didier Buchs

Université de Genève

15 mars 2018

Typage et règles de visibilité

- Typage, sous-typage
- Typage faible, fort, statique, dynamique
- Règles de visibilité
- Durée de vie des variables

Typage et règles de visibilité : La notion de type

Un type est défini par :

- un ensemble de valeurs possibles
- les opérations portant sur ces valeurs

Les types permettent de caractériser les données manipulées par le programme ; ce sont des contraintes

- pour fixer l'interprétation des bits en mémoire :
  - 01000001 = 65 (vu comme un entier) ou 'A' (comme caractère)
- pour s'assurer qu'opérations et valeurs aient un sens :
  - dépassement de capacité : 65535+1 = 0
  - opération incompatible : log (true)
- pour différencier des valeurs logiquement distinctes :
  - pommes $\neq$ *poires*

La notion de type (suite)

- Les types doivent être aussi proches que possibles des concepts du monde réel (le problème à résoudre) : *ils sont un support pour l'abstraction*
- Les types expriment des contraintes qui sont contrôlées par le langage (*statiquement ou dynamiquement*) ; elles permettent ainsi d'automatiser la vérification de la cohérence du logiciel
  - contraintes au service du programmeur !
- Les violations de type constituent des erreurs de nature sémantique (statique ou dynamique)

Erreurs sémantiques statiques vs. dynamiques

```
PROGRAM test;
VAR
  bool : boolean;
  entier : integer;
  petit : 0..32;
BEGIN
  bool := entier;          Erreur de typage statique
  entier := 34;            OK
  petit := entier;         Erreur de typage dynamique
END.
```

Avant la dernière affectation, le compilateur Pascal aura généré le code de vérification suivant :

```
IF (entier < 0) OR (entier > 32) THEN
  BEGIN
      writeln('Fatal: dépassement de capacité');
      halt;
  END;
```

La notion de type (suite)

- Un nom est en général attribué à chaque type
- Dans les langages évolués, toute valeur a un type
  - mais dans les langages de bas niveau, presque tout se confond en une notion de mot-machine (chaîne de bits, de taille propre à la machine sous-jacente)
  - type atomique : valeurs indécomposables et occupant au maximum un mot-machine (entiers, booléens)
- Certains types sont prédéfinis (dépendant du langage) :
  - Entier, caractère : pratiquement universels
  - Nombre réel, vecteur : langages algorithmiques
  - Liste : langages de l'IA (lisp, prolog)
  - Fonction, objet : langages fonctionnels, resp. à objets

Types simples

- Les types simples (ou scalaires) comprennent :
  - les types discrets (ou ordinaux), dont les valeurs peuvent être énumérées :
  - entiers { . . .,-1,0,1,2,. . .}
  - booléens { vrai, faux }
  - caractères { . . .,' ?','@','A','B',... }
  - intervalles { 0..limite sup }
  - énumérés { lundi, mardi, . . . }
- les types réels :
  - à virgule flottante : 2.7777777778E-10
  - à virgule fixe (les rationnels de Ada) :
    - type Volt is delta 0.125 range 0.0 .. 255.0 ;

Types simples (suite)

- Les types atomiques sont un sous-ensemble des types simples
  - Un type atomique est un type de donnée manipulable en une instruction machine indivisible, et dont la valeur sera toujours cohérente même si le processus courant est provisoirement interrompu.
  - Typiquement : le mot-machine (toute entité de taille supérieure ou inférieure risque un traitement en plusieurs étapes)
- Les types simples prédéfinis sont parfois dits primitifs
  - Mais les types primitifs désignent parfois tous les types prédéfinis du langage considéré, y compris des types structurés tels que les chaînes de caractères. . .

Types structurés

- Par opposition, un type structuré est une composition d'éléments de type simple :
  - enregistrements / articles (record, class ou struct) TYPE z = RECORD re, im : real END ;
  - enregistrements à variantes (union, record .. case) Voir l'exemple de la page suivante
  - vecteurs (array)
  - ensembles (set)
  - listes ([*tete*|*queue*] en prolog, car et cdr en lisp)
  - fichiers (file)
- Les mots-clés tels que 'record', 'array' ou 'file' sont des constructeurs de types qui engendrent de nouveaux types

Types structurés

Enregistrement à variantes :

```
type Figure (Genre: (Triangle, Carre) := Carre) is
   record
      couleur : TypeCouleur := Rouge ;
        case Genre of
        when Triangle =>
            pt1,pt2,pt3 : Point;
        when Carre =>
            supgauche : Point;
            longueur : INTEGER;
        end case;
  end record;
```

- La même zone mémoire sert à stocker les différentes alternatives : superposition des variantes (mutuellement exclusives), d'où gain de place
- Le discriminant (ici : Genre) est considéré comme un champ supplémentaire, commun à toutes les variantes

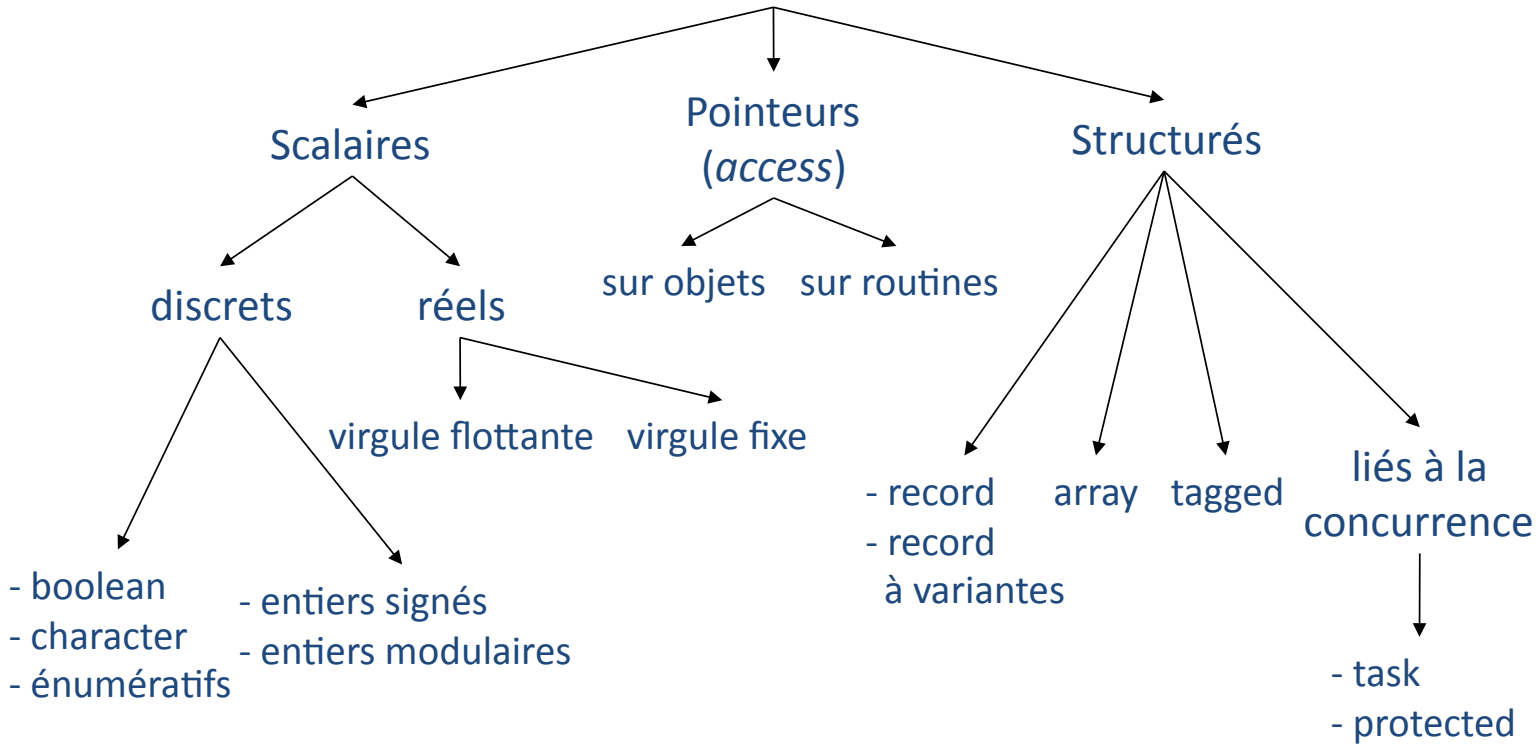
Enregistrement à variantes (suite)

En Ada, une instruction proc(fig.longueur) sera complétée par le compilateur avec un test tel que :

```
if fig.Genre = Carre then
        proc(fig.longueur);  -- tout est OK
    else
        raise constraint_error;  -- signaler l'erreur !
end if;
```

- En Pascal et C (mais pas Ada) : conversions brutes possibles entre alternatives (en l'absence de discriminant), d'où risque d'erreurs sémantiques et même de corruption de la mémoire.
- La programmation par objets a rendu ce type d'approche assez désuète (sauf en programmation système)

Hiérarchie de types du langage Ada95



Typage faible, fort

- Un langage est dit à typage faible s'il n'effectue pas (ou peu) de vérifications
  - Le langage C est très laxiste :

```
int i; float f; char c;

i = f*c;
```

- Un langage est dit à typage fort s'il exige que les vérifications soient faites systématiquement
  - Java, Ada, Haskell
  - Pascal (dans une moindre mesure)

Typage statique, dynamique

- Un langage est dit à typage statique s'il est fortement typé et s'il effectue (l'essentiel de) ses tests à la compilation
  - A l'extrême, la notion de type peut donc être une notion qui n'existe que dans le compilateur, et qui ne laisse pas de trace dans l'implémentation
  - Représentants principaux, bien que beaucoup de tests se fassent aussi à l'exécution : Ada,Scala,Swift
- Un langage à typage dynamique effectue les contrôles (uniquement ou essentiellement) à l'exécution
  - Lisp, Prolog, Smalltalk, Python
  - Tous les langages à portée dynamique
  - Considéré peu sûr pour les grands projets

Contrôle du typage

- Pour effectuer les contrôles de typage, les langages disposent de règles : équivalence de type, compatibilité de type et inférence de type
- L'équivalence de type définit si deux types sont considérés identiques
  - équivalence structurelle : on se base sur la représentation interne (Modula-3, ML)
  - équivalence de nom : c'est le nom du type qui est déterminant (Ada, Java). Cette approche est préférée actuellement, car elle supporte l'abstraction : deux types logiquement distincts sont considérés différents même si leur représentation interne est identique
    - P.ex : coordonnées cartésiennes et polaires

Compatibilité de type

- La compatibilité de type détermine quand un objet d'un certain type peut être utilisé dans un certain contexte
  - S'il y a équivalence entre le type de l'objet et le type attendu par le contexte, pas de problème
  - Sinon, l'usage est légal si
    - il est possible d'effectuer une coercion (ou conversion implicite) de type
      Exemple en Pascal :

      ```
      i : integer; j : 0..10;
      i := j;
      ```

    - il y a un forçage (ou conversion explicite) de type
      Exemple Ada avec conversion et test à l'exécution :

      ```
      n : integer; r : real;
      n := integer(r);    -- overflow ?
      ```

Inférence de type

- L'inférence de type consiste pour le compilateur à déterminer, à partir du type des sous-expressions, le type résultant pour l'ensemble de l'expression analysée
- Polymorphisme : indique qu'une valeur peut-être de plusieurs types
  exemple : 2 :nat et 2 :int
- Difficile en présence d'opérateurs génériques ou surchargés, et de conversions implicites permises
  - exemple en C++ : pour la 3ème instruction, le compilateur doit déterminer qu'il y a coercion de i en float (nombre réel), puis reconversion du résultat de la multiplication en int (nombre entier).

```
int i = 8 ;
float k = 3;
int r = i*k;
```

Sémantique du typage statique

catégories syntaxiques :

$$e, e', e'' \in ExpVar, op \in Op, n \in Num, v \in VarNum$$

$$op ::= + | - | * | div$$

$$e ::= v | n | e' \ op \ e''$$

On ajoute à chaque expression un type
Une expression e de type t se note (: est un prédicat) :

$$e : t$$

$$1 + 2 : nat$$

Comment calculer le type d'une expression ?

Règles de typage, inférence de type

- règles de typage simple :
  $\frac{}{n:nat}$ $\frac{e':t, e'':t}{e' \ op \ e'':t}$
  par exemple : $2 : nat$, $1 + 2 : nat$
- problème si t peut être *nat* ou *integer* ?
  $(2 - 3) : nat$?
  $(2 - 3) : integer$?
- régles de typage avec polymorphisme ad-hoc
  n :nat

  $\frac{}{(-n):integer}$
  $\frac{e':t, e'':t}{e' \ + \ e'':t}$ idem pour * et /
  $\frac{e':t, e'':t}{e' \ - \ e'':integer}$
- cohercion
  $\frac{e:nat}{e:integer}$
  $2 + (-3) : integer$!

Règles d'inférence de type

- Règle pour type simple : Integer,natural, boolean
- Règle pour type composés : Listes, tableaux ... $\frac{e':t, e'':t'}{e'::e'':t, t' \rightarrow t';}$
- Règle pour type composé générique (v : variable de type)
$\frac{e':v, e'':t'}{e'::e'':v, t' \rightarrow t';}$

Exercice

Donner une def sémantique typée (pour expressions)

Sous-typage

- Dans certains langages, le programmeur peut définir des sous-types d'un type défini au préalable par lui ou par le langage
- Si S est un sous-type de T, alors une expression de type S peut utilisée partout où il est légal d'en utiliser une de type T ; une conversion de type implicite prend alors place, il s'agit du principe de substitutivité.
- Le sous-typage est largement employé dans les langages orientés objets
- Polymorphisme : Si $a : A$ et $A < B$ alors $a : B$

$$\frac{e:t, t<t'}{e:t'}$$

Sous-typage (suite)

Un sous-type est souvent un type dont le domaine de valeurs est contraint par rapport à celui du super-type
Exemple Ada :

```
TYPE Jours IS  (lu, ma, me, je, ve, sa, di);
SUBTYPE Weekend IS Jours RANGE sa..di;
J : Jours;
W : Weekend;
...
J := W;  -- pas de problème
W := J;  -- exception si J n'est ni 'sa' ni 'di'
```

Genres de sous-types

- Restriction d'ensemble, comme précédement en Ada
  $nat < integer$
- Spécialisation :
  $colorprinter < printer$
  Des informations supplémentaires sont fournies 'spécialisant' le type, par exemple dans cet exemple la couleur d'impression.

Qu'en est-il des opérations des sous types ?

- Les opérations doivent également suivrent une relation de préordre !
- L'arité des opérations est le type du profil : $t_1, t_2, \ldots, t_n -> t$
  - Exemple :
    $+ : integer, integer -> integer; + : nat, nat -> nat;$
- Si $t < t'$ qu'en est-il du profil des opérations de t respectivement t'
- Deux types de relations :
  - Co-variante : les types entre domaine et co-domaine évoluent dans le même sens.
  - Contra-variante : les types entre domaine et co-domaine évoluent dans le sens contraire.

Subclassing et subtyping

- On parle de subclassing plutôt que de subtyping lorsque le principe de substitutabilité d'éléments d'une classe par des éléments d'une autre ne conduit pas à la préservation du comportement.
- Sauf pour des contraintes simples (exemple Ada précédent sur la réduction d'intervale de valeurs) les langages de programmations ne garantissent pas ce principe.
- Subclassing est lié à l'héritage de code depuis une classe parente.

Types abstraits

- Un type abstrait est un type dont la définition n'est pas rendue publique : principe d'encapsulation
  - Le but est de cacher l'implémentation, de manière à pouvoir facilement la changer plus tard si nécessaire
  - L'accès au contenu d'un type abstrait ne se fait qu'indirectement à travers des fonctions et des procédures (les accesseurs)
  - Ceci diminue la dépendance du code vis-à-vis de la représentation interne, donc des détails d'implémentation
- Exemple d'utilisation d'un type abstrait, qui fonctionnera toujours, que 'lePoint' soit implémenté en coordonnées cartésiennes ou polaires : translater(lePoint, dx, dy)

Fonction d'ordre supérieur et typage

Comment donner une sémantique avec des fonctions d'ordres supérieurs

- Théorie du $\lambda - calcul$ (Alonzo Church)

Comment coder les types de données connus

- Théorie du codage (Alonzo Church et Dana Scott)

Et des centaines d'extensions pour décrire à peu près toutes les évolutions des langages, en particulier les notions de typage.

Fonction d'ordre supérieur : termes

Definition (Syntaxe langage fonctionnel)

$M, N ::= x|(\lambda x.M)|(MN)$

- $fv(x) = \{x\}$
- $fv(MN) = fv(M) \cup fv(N)$
- $fv(\lambda y.M) = fv(M)/\{y\}$

- $bv(x) = \emptyset$
- $bv(MN) = bv(M) \cup bv(N)$
- $bv(\lambda y.M) = bv(M) \cup \{y\}$

We write $x \notin M$ for $x \notin fv(M) \cup bv(M)$.

Fonction d'ordre supérieur : substitution

- $x[N/x] = N$
- $y[N/x] = y$, si $y \neq x$
- $(PQ)[N/x] = P[N/x]Q[N/x]$
- $(\lambda y.M)[N/x] = \lambda y.M$, si $y = x$
- $(\lambda y.M)[N/x] = \lambda y.(M[N/x])$, si $y \neq x \wedge y \notin fv(N)$
- $(\lambda y.M)[N/x] = \lambda z.(M[z/y][N/x])$, si $y \neq x \wedge y \in fv(N) \wedge z \notin fv(NM)$

Fonction d'ordre supérieur : α conversion

Les termes sont equivalent modulo renommage.

$$(\lambda y.M) =_\alpha \lambda z.M[z/y], \text{ si } z \notin M$$

En general on considère les termes modulo renommage (si on renomme l'évaluation donne le même résultat)

Fonction d'ordre supérieur : β reduction

Les termes sont reductible.

$$(\lambda y.M)N \rightarrow_\beta M[N/y]$$

En general on considère les termes modulo renommage.

La semantique du lambda calcul est alors la fermeture transitive de la relation $\rightarrow_\beta$ et l'obtention d'une forme irréductible.

Encodage des types de données :entiers

- $0 = \lambda f.\lambda x.x$
- $1 = \lambda f.\lambda x.fx$
- $2 = \lambda f.\lambda x.f(fx))$
- ...

- $succ = \lambda n.\lambda f.\lambda x.f(nfx)$
- $+ = \lambda m.\lambda n.\lambda f.\lambda x.m(f(nfx))$
- $* = \lambda m.\lambda n.\lambda f.\lambda x.m(nf)x$

Evaluation de *succ* 2

1. $succ\ 2 = \lambda n.\lambda f.\lambda x.f(nfx)(\lambda f.\lambda x.f(fx))$
2. $succ\ 2 = \lambda f.\lambda x.f\lambda f.\lambda x.f(fx)fx$
3. $succ\ 2 = \lambda x.\lambda f.\lambda x.f(fx)fx$
4. $succ\ 2 = \lambda f.\lambda x.f(f(fx)) = 3$

Encodage des types de données : booleens

- $true = \lambda a.\lambda b.a$
- $false = \lambda a.\lambda b.b$

- $and = \lambda p.\lambda q.pqp$
- $or = \lambda p.\lambda q.ppq$
- $not = \lambda p.p(\lambda a.\lambda b.b)(\lambda a.\lambda b.a)$

Extension du lambda calcul : let

Les termes sont enrichis de : $let\, x\, =\, M\, in\, N$

$$let\, x\, =\, M\, in\, N \rightarrow_{\beta} N[M/x]$$

sucre syntaxique facilitant la lecture et l'evaluation en 'sequence' des termes.

Système de types : Hindley- Milner

jugement de type :

$$\Gamma \vdash \Delta$$

avec des variables $x_1, x_2, ..., x_n$ et des termes t_1, t_2, t_n et un ensemble de types $\tau_1, \tau_2, ..., \tau_n \in T$

$$t_1 : \tau_1, ..., t_n : \tau_n \vdash \Delta$$

$$\frac{}{\Gamma, x : \tau \vdash \Delta, x : \tau}(Var)$$

$$\frac{\Gamma \vdash M : \tau \to \tau', \Gamma \vdash N : \tau}{\Gamma \vdash \Delta, MN : \tau'}(App)$$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \Delta, \lambda x.M : \tau \to \tau'}(Abs)$$

Constructions suplémentaires

let et genericité

$$\frac{\Gamma \vdash M : \tau, \Gamma, x : \tau \vdash N : \tau'}{\Gamma \vdash \Delta, let\ x\ =\ M\ in\ N : \tau'}(Let)$$

$$\frac{\Gamma \vdash M : \tau, \alpha \notin free(\Gamma)}{\Gamma \vdash \Delta, M : \forall \alpha.\tau}(Gen)$$

Le type construit doit être le plus abstrait. (le moins de variables instanciées.)
Un ensemble de types de base est généralement proposés, tels que *int*,*bool*,etc. avec les opérations associées.
Un algotithme est implémenté (algorithme W) pour construire un système de types suivant ces règles.

Identificateurs dans les langages informatiques

- Le nommage est aussi un support pour construire l'abstraction de manière incrémentale
- Les règles varient d'un langage à l'autre
  - syntaxe
  - possibilité d'alias ou d'homonymes
  - durée de vie
  - visibilité

Qui a besoin d'un nom ?

- Un nom est un moyen de se référer aux différentes entités d'un programme. On désigne quelque chose par un nom si on veut créer cette chose, l'utiliser, la changer ou la détruire.
- Les entités qui ont besoin d'un nom sont :
  - Les constantes et les variables,
  - Les opérateurs,
  - Les labels,
  - Les types,
  - Les procédures et les fonctions
  - Les modules et les programmes,
  - Les fichiers et les disques,
  - Les commandes et les items de menus,
  - Les ordinateurs, les réseaux et les usagers

Liaison (en anglais : binding)

- La liaison est l'action d'associer une signification à un symbole ; elle a lieu :
  - à la définition du langage (signification des mots-clés)
  - à la création du compilateur et des librairies de support
  - dépendances p.r. au système et au matériel
  - à l'écriture et la compilation de l'application
  - à l'édition de liens
  - au chargement (dynamique)
  - à l'exécution (dynamique)
- Plus les liaisons s'établissent tôt, plus l'application s'exécutera efficacement, mais ce sera au détriment de la flexibilité

Variables

- Une variable est une abstraction de cellule de mémoire ou d'une collection de cellules.
- Une variable peut être caractérisée par un 6-tuplet : (Nom, adresse, valeur, type, durée de vie, portée)
- L'usager décide du nom et du type de la variable. L'emplacement de la déclaration décide de sa portée et de sa longevité. Son addresse est déterminée pendant l'exécution et sa valeur depend des instructions dans lesquelles elle apparaît.

Exemple de liaisons de variables

Attribut	Moment de détermination	Cause
nom	Compile time	déclarations
addresse	Load time ou run time (Pascal)	phénomène implicite
type	Compile time (Pascal) run time (Smalltalk)	déclarations
valeur	Run time ou load time (initialization)	Instructions surtout affectation
durée de vie	Compile time	Déclarations
portée	Compile time	Placement des déclarations

Portée, visibilité, durée de vie

- La portée d'un identificateur correspond aux parties du programme où cet identificateur a une liaison (donc où il réfère au même 'objet')
- La visibilité d'un identificateur correspond aux endroits où cet identificateur peut être utilisé
  - un identificateur peut être invisible à l'intérieur de sa portée, s'il est temporairement caché par un homonyme
- La durée de vie d'un objet en mémoire (en particulier d'une variable) correspond à la période allant de sa création (allocation de mémoire) à sa destruction (libération de cette mémoire), en passant par sa (ses) portée(s).
  - Plusieurs portées en cas d'alias
  - Portée $>$ durée de vie $=$ erreur de programmation !

A propos de la durée de vie

- L'allocation de mémoire pour un objet se fait ou bien au chargement (load-time) ou à l'exécution (run-time)
- Il existe donc deux classes de variables :
  - Les variables statiques : l'allocation est faite une seule fois, avant que le programme commence son exécution
    - variables globales
    - variables locales rémanentes (static en C)
  - Les variables dynamiques : l'allocation est faite pendant l'exécution du programme
    - allocation explicite : dans le tas (avec new, malloc)
    - allocation implicite : sur la pile (variables locales automatiques)

Portée

- Le concept de bloc est un des fondements de la programmation structurée Déclarations et instructions sont groupées en blocs afin de :
  - Grouper les étapes d'une instruction non-élémentaire
  - Interpréter les noms adéquatement.
- Les blocs peuvent parfois être emboîtés. Les noms introduits dans un bloc s'appellent les liaisons locales. Un nom mentionné mais pas défini dans un bloc doit avoir été défini dans l'un des blocs de l'englobants.

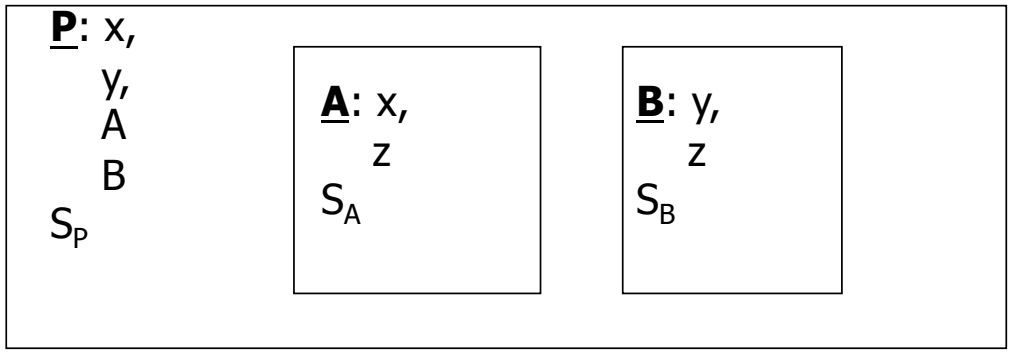
Blocs anonymes et blocs nommés

- Un programme, une procédure ou une fonction sont des exemples de blocs nommés.
- Un bloc anonyme est comme une procédure sans nom qui est appelée immédiatement lors de son occurrence dans le texte du programme.
- Les blocs anonymes sont utiles lorsqu'un calcul est nécessaire une seule fois et que ses variables ne sont utiles que dans ce calcul : on ne veut pas les déclarer à l'extérieur de ce bloc.
  - Les blocs anonymes existent p.ex. en Ada et en C, mais en C, les blocs nommés imbriqués ne sont pas permis

Visibilité, masquage d'identificateurs

Si le même nom X est défini dans dans un bloc environnant B1 et dans un bloc emboîté B2 (d'où $B2 \subset B1$), alors la visibilité du X défini en B1 est perdue dans le bloc B2 qui ne voit que le X défini en B2.

Exemple :



Portée statique et dynamique

- Dans les langages à portée statique (ou lexicale), la portée des identificateurs est fixée à la compilation de manière à être le plus petit bloc englobant leur déclaration.
- La portée dynamique cherche l'identificateur dans la chaîne des procédures appelantes. Cette chaîne considère les règles de visibilité mais pas l'emboîtement.
- La portée dynamique a surtout l'avantage d'être plus simple à implémenter : pas de notion de lien statique (pointeur sur le bloc d'activation du bloc englobant) en plus du lien dynamique (pointeur sur le bloc d'activation de l'appelant)

Exercice

```
PROGRAM P;
VAR X: integer;
  PROCEDURE A;
  BEGIN
    X:= X+1;     print(X);
  END;
  PROCEDURE B;
  VAR X:integer;
  BEGIN
    X:= 17;     A;
  END;
BEGIN (* P *)
  X:= 23;    B;
END;
```

Quel résultat nous donnerait ce programe pour :

- portée statique
- portée dynamique

Surcharge

- La surcharge (ou polymorphisme ad-hoc ; overloading en anglais) consiste à donner plusieurs significations concurrentes à un même symbole (en général identificateur de fonction ou procédure)
- Les opérateurs mathématiques sont surchargés par défaut dans la plupart des langages
  - le $-$ peut être monadique ou dyadique
  - le $+$ opère sur les entiers et les réels
- Ada, C++ et Haskell permettent au programmeur de définir des opérations surchargées ;
  - le compilateur exigera de pouvoir les distinguer par leur profil (nombre et type des arguments, y compris un éventuel paramètre résultat)

Surcharge (suite)

- Pratique pour signifier intuitivement une sémantique similaire à d'autres identificateurs identiques connus
  - La surcharge est un sucre syntaxique : ce n'est qu'un raccourci pour le confort du programmeur
- Mais l'abus peut avoir un effet inverse
- Impact négatif sur l'interopérabilité : la surcharge rend difficile l'exportation de définitions vers des programmes écrits dans d'autres langages

Conclusion

- Typage : vaste sujet de réflexion, de nombreuses approches théoriques et pratiques.
- Typage : compromis entre detection statique et dynamique des erreurs
- Portée des variables et en général des identificateurs est importante, sa non-maîtrise implique de nombreuses erreurs
  - 'dangling references'
  - 'aliasing'
  - 'memory leak'