Langages et paradigmes

Olivier Lecarme

Master d'Informatique, première année

2007–2008

Deuxième partie II

Noms

Généralités sur les noms

Les langages dits « de haut niveau » tirent leur nom du niveau d'abstraction qu'ils fournissent. Il s'agit de s'éloigner autant que possible des détails de l'ordinateur, mais aussi de faciliter la programmation.
Un nom sert à désigner autre chose, et en particulier une abstraction, que ce soit des données ou des actions.

Moment de prise des décisions

- à la conception du langage : choix des structures d'énoncés, des constructeurs de types
- à la construction de l'implémentation : choix du domaine des types de base, de la précision des nombres, des limites variées
- à l'écriture du programme : choix des algorithmes, des structures de données, des noms
- à la compilation du programme : correspondance entre les structures de haut niveau et la machine

moment de prise des décisions

- à l'édition de liens : résolution des références entre les modules
- au chargement : choix des adresses en mémoire, réelle ou virtuelle
- à l'exécution : valeurs des variables

On appelle en général statiques les choix faits avant l'exécution, dynamiques ceux qui sont faits à l'exécution. Plus le choix est fait tardivement, plus la souplesse d'utilisation est grande, mais moins on peut prédire ce qui va se passer. Par conséquent, les choix dynamiques impliquent de moins bonnes performances.

Durée de vie

- Un *nom* désigne un *objet* grâce à une *liaison*. Cela implique les événements suivants :
  - création des objets
  - création des liaisons
  - références aux objets grâce aux liaisons
  - suspension et réactivation de certaines liaisons
  - destruction des liaisons
  - destruction des objets

durée de vie

- L'intervalle entre la création et la destruction de la liaison entre un nom et un objet est sa *durée de vie*. Même chose pour l'objet lui-même.
- Les deux durées de vie ne coïncident pas forcément.
- Les durées de vie des objets correspondent à trois grandes catégories de gestion de l'espace

Gestion de l'espace

- allocation *statique* :
  - les objets ont une adresse fixée pour la durée du programme
  - variables globales
  - variables rémanentes
  - constantes

gestion de l'espace

- allocation *en pile* :
  - les objets sont détruits en ordre inverse de leur création
  - cela se fait en parallèle avec les appels et retours de sous-programmes
  - dans un langage sans récursivité, les variables locales aux sous-programmes peuvent être allouées statiquement
- allocation *en tas* :
  - les objets sont créés et détruits à des moments arbitraires
  - cela implique en général une gestion élaborée de l'espace

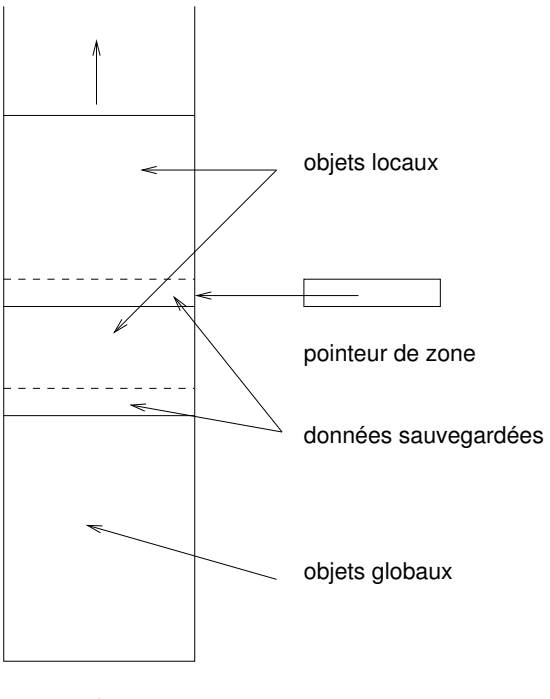
Allocation en pile

- chaque sous-programme est associé à une *zone d'activation*
- les zones d'activation sont toutes placées dans la *pile d'exécution*
- à l'intérieur de la zone, les adresses relatives des objets peuvent être connues à la compilation
- un *pointeur de zone* sert d'adresse de base aux objets locaux

allocation en pile

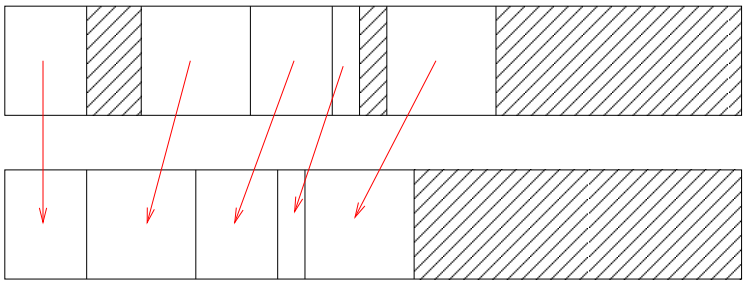
- l'adresse de retour et les contenus des registres sont également sauvegardés dans la zone d'activation
- le *prologue* et l'*épilogue* du sous-programme gèrent l'allocation, la gestion et la libération de la zone

allocation en pile



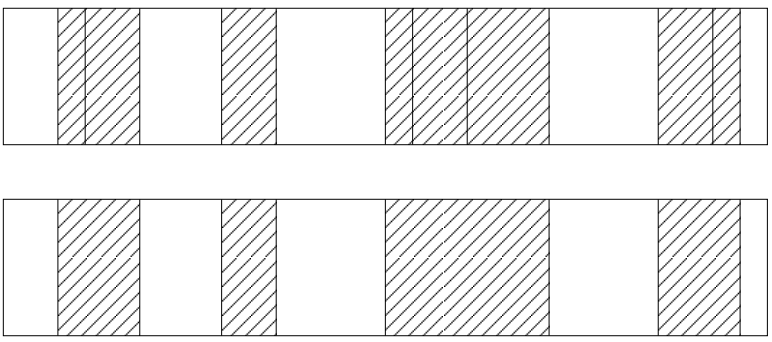
Allocation en tas

- dans le tas les blocs de mémoire sont alloués et libérés de manière arbitraire
- après un certain nombre d'allocations et de libérations, le tas est *fragmenté* et doit être compacté
- les techniques de récupération de mémoire déterminent quels blocs sont encore occupés pour les regrouper de manière contiguë



allocation en tas

- il existe aussi des techniques qui acceptent une certaine fragmentation et regroupent les blocs libérés s'ils sont contigus
- la récupération de mémoire classique peut bloquer temporairement l'exécution du programme
- certains langages imposent la libération explicite des blocs, ce qui évite l'utilisation d'un récupérateur



Règles de portée

La portée d'une liaison est la zone textuelle du programme où elle est valide.

Portée statique

- Dans la plupart des langages, la portée est déterminée de manière statique.
- Une liaison est temporairement désactivée si l'on entre dans un sous-programme où une autre partie de programme qui en crée une portant le même nom. Elle est réactivée à la fin de cette partie.
- On parle de portée *statique* ou *lexicale*.

Différences entre les langages

- Cobol, Basic à sa définition : une seule portée statique pour tout le programme, tous les noms doivent être distincts.
- Fortran : une portée par sous-programme, tous les sous-programmes sont disjoints, *blocs communs* pour définir des portées importées de manière sélective par les sous-programmes.

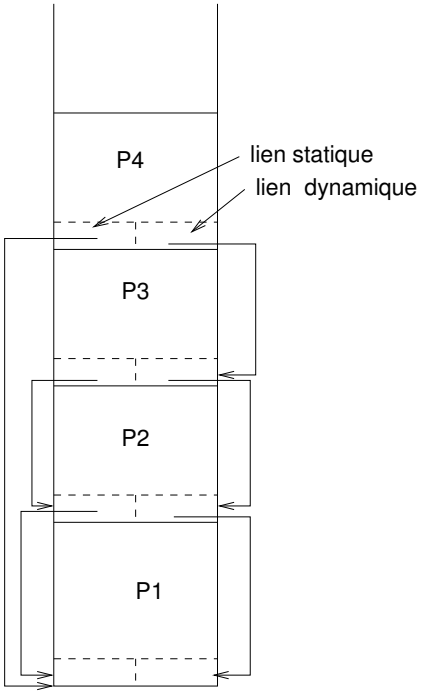
différences entre les langages

- C, Java, Icon : pas d'emboîtement des sous-programmes, donc une portée générale statique pour les noms globaux, et masquage de ces noms par les noms homonymes dans les sous-programmes
- Algol, Pascal, PL/I, Ada, etc. : emboîtement des sous-programmes, qui fait gérer la portée statique selon un mécanisme de pile

portée statique

- La portée dans les sous-programmes emboîtés est gérée à l'exécution par un *lien statique*, qui permet de savoir pour chaque zone d'activation quelle est la zone du sous-programme emboîtant.
- Problème : quand commence la portée d'une liaison ? À la déclaration du nom (Ada), ou au début du sous-programme (Pascal) ?

Exemple de sous-programmes emboîtés




```
procedure P1(A1 : T1) ;
var X : real ;
    procedure P2(A2 : T2) ;
        procedure P3(A3 : T3) ;
        begin ... end ;
    ...
    begin ... end ;
    procedure P4(A4 : T4) ;
        function F1(A5 : T5) : T6 ;
        var X : integer ;
        begin ... end ;
    ...
    begin ... end ;
begin ... end ;
```

Modularité

- Un *module* sert à masquer les détails d'implémentation.
- La portée des liaisons ne peut pas être automatique.
- Il faut donc des déclarations explicites d'*importation* et *exportation*.
- Variations importantes suivant les langages : Modula-2, Ada, C^{++} par exemple.

modularité

- En-dehors du module, les liaisons internes au module sont inactives mais non pas détruites.
- L'importation réactive la liaison, éventuellement avec des restrictions imposées par le module (liaison opaque, privée, protégée, etc.).

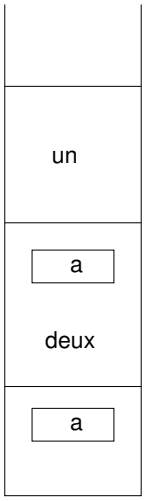
Portée dynamique

- Les liaisons entre noms et objets dépendent du déroulement de l'exécution et de l'emboîtement des *appels* de sous-programmes.
- Les langages à portée dynamique sont hors de l'ordinaire :
  - APL
  - Snobol4
  - Anciens dialectes de Lisp
  - Perl (au choix du programmeur)
  - Les shells

portée dynamique

- Le mécanisme se comprend facilement par son implémentation : les noms sont empilés à l'entrée dans les sous-programmes, et retirés à la sortie ; la recherche d'une liaison se fait à partir du sommet de la pile.

Exemple de portée dynamique




```
var a : integer ; ...
procedure un ;
begin a := 1
end ;
procedure deux ;
    var a : integer ;
    begin un
    end ;
begin
    a := 2 ;
    if lirentier > 0 then deux
    else un
    endif
    ecrirentier(a)
end ;
```

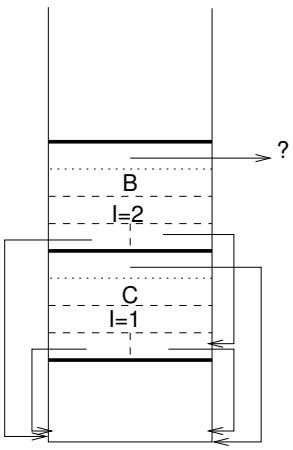

Liaison des environnements

- Fermeture de sous-programme
  - Que se passe-t-il quand un sous-programme peut lui-même être manipulé comme un objet, ou quand on peut avoir des références à des sous-programmes ?
  - La difficulté est que la structure textuelle du programme n'est plus significative, puisque le sous-programme peut être appelé à un endroit où il n'est pas directement visible.

fermeture de sous-programme

- La portée dyamique conduit à une règle de *liaison de surface*, et la portée statique à une règle de *liaison en profondeur*.
- L'implémentation construit une *fermeture* du sous-programme, qui contient un pointeur sur le code et le lien statique.

Exemple




```
program exemple ;
    procedure A(I : integer ; procedure P) ;
        procedure B ;
        begin
            writeln(I)
        end ;
    begin
        if I>1 then P
        else A(2, B)
        endif
    end ;
    procedure C ;
    begin end ;
begin
    A(1, C)
end.
```

Classe des sous-programmes

- Classes des valeurs :
  - Une valeur est de *première classe* si on peut la passer comme paramètre, l'affecter à une variable ou la fournir comme résultat d'une fonction.
  - Une valeur de *deuxième classe* peut être passée comme paramètre, mais c'est tout.
  - Une valeur de *troisième classe* ne peut pas être manipulée.

Que font les langages ?

- les étiquettes sont de troisième classe dans presque tous les langages, mais de deuxième classe en Algol 60
- les sous-programmes sont de deuxième classe dans la plupart des langages procéduraux, de troisième classe dans la première définition d'Ada

que font les langages ?

- les sous-programmes sont de première classe dans tous les langages fonctionnels
- avec certaines restrictions, également en Modula-2, Ada 95, C et C^{++}.
- seuls quelques langages fonctionnels permettent de créer de nouveaux sous-programmes à l'exécution.

Surcharge

- Un nom qui peut se référer à plusieurs objets distincts dans une portée donnée est *surchargé*.
- Si au contraire plusieurs noms se réfèrent au même objet, on parle d'*alias*.
- Presque tous les langages surchargent au moins les opérateurs arithmétiques les plus simples.

surcharge

- Ada permet de surcharger les constantes énumérées.
- Ada, C^{++} et Java permettent de surcharger les noms de sous-programmes, à condition que le profil des sous-programmes permette de les distinguer.
- Algol 68, Ada, C^{++} et Fortran 90 permettent de surcharger les opérateurs, dans les mêmes conditions.

surcharge

- À ne pas confondre avec
  - conversion automatique : un paramètre d'un type inattendu est automatiquement converti dans le type demandé (Fortran, C, Pascal)
  - polymorphisme : plusieurs types distincts sont acceptés pour le même paramètre (langages fonctionnels, C^{++}, Java)
  - Pascal et Modula-2 fournissent un polymorphisme réduit avec les tableaux *conformants*

généricité

- généricité : le sous-programme est un modèle qui permet de créér plusieurs sous-programmes concrets, suivant la valeur d'un paramètre de généricité qui peut être un type par exemple (Ada)

Difficultés avec les portées

Trois exemples pris dans Pascal, qui a pourtant des règles simples :

- Redéfinition du résultat d'une fonction :

```
function toto : integer;
    ...
    type toto = ...
    begin
        ...
        toto := 10
```

difficultés

- Utilisation d'un nom avant sa redéfinition :

```
const toto = -1 ;
procedure lili ;
    const jojo = toto ;
        toto = 0 ;
```


difficultés

- Autre cas plus compliqué :

```
procedure toto ;
begin
    ...
end ;
procedure jojo ;
    procedure lili ;
        begin
            toto
        end ;
    procedure toto ;
```

difficultés

- Cas des types récursifs :

```
type
    ptrA = ^A ;
    A = record
        suiv : prtA ;
        ...
    end ;
```