Infrastructure d'exécution

Sections 7.1 et 7.2

* Contenu *

- Procédures
  - Arbre d'activation
  - Pile de contrôle
  - Déclarations (liaison de nom et portée)
  - Organisation en mémoire
  - Structure d'activation
- Stratégies d'allocation de la mémoire
  - Statique
  - Par pile
  - Par tas

Introduction

Chapitre 7

Ce chapitre se concentre sur le lien entre le programme source et les actions qui se produiront à l'exécution du programme, hormis le code et sa génération.

En particulier, nous étudions le lien entre les *identificateurs* et les *variables* qui servent à contenir leur valeur.

L'allocation et la désallocation des données sont gérées par le module d'*infrastructure d'exécution*.

À chaque fois qu'une procédure (ou fonction) est exécutée, on parle d'une *activation* de la procédure.

Dans le cas de procédures récursives, plusieurs activations de la même procédure peuvent coexister.

Procédures

Nous reprenons quelques définitions.

Une *définition de procédure* est une déclaration qui associe un identificateur à un énoncé.

L'identificateur est le *nom* de la procédure. L'énoncé est le *corps* de la procédure.

Suivant le vocabulaire de certains langages, une procédure qui retourne une valeur est une *fonction*.

On dispose d'une notation (comme '`f(a, b, c)`') servant à indiquer que la procédure est *appelée*.

Certains identificateurs inclus dans la définition de procédure ont un rôle spécial et sont appelés *paramètres formels*.

Des *arguments*, ou *paramètres actuels*, peuvent être passés à la procédure lors de l'appel. Ils "remplacent" les paramètres formels de la procédure au cours de cet appel.

Arbres d'activation

Le *flot de contrôle* est une suite de pas dans le code d'un programme, chaque pas indiquant l'énoncé à exécuter ou l'expression à évaluer.

On fait les hypothèses suivantes à propos du *flot de contrôle*:

- Le flot de contrôle se déplace séquentiellement.
- L'exécution d'une procédure commence au début du corps de la procédure et rend éventuellement le contrôle au point qui suit immédiatement l'endroit d'où s'est fait l'appel.

La *durée de vie* de l'activation d'une procédure 'p' est la séquence des pas entre le premier et le dernier pas de l'exécution du corps de la procédure, inclusivement.

Propriété: Si a et b sont des activations de procédures, alors leurs durées de vie sont soit disjointes, soit imbriquées.

Une procédure est *récursive* si une nouvelle activation de la procédure peut débuter avant la terminaison d'une activation ayant débuté auparavant. La récursion n'a pas à être directe.

Arbres d'activation

On peut présenter graphiquement le flot de contrôle d'une exécution qui implique des procédures à l'aide d'un *arbre d'activation*, chez lequel:

- chaque noeud représente une activation d'une procédure;
- la racine représente l'activation du programme principal;
- le noeud d'une activation a est le parent d'une activation b si et seulement si le contrôle passe [directement] de a à b;
- le noeud associé à a se situe à la gauche du noeud associé à b si et seulement si la durée de vie de a a lieu avant la durée de vie de b.

Arbres d'activation

Exemple 7.1. Voici le pseudo-code d'un algorithme QuickSort:

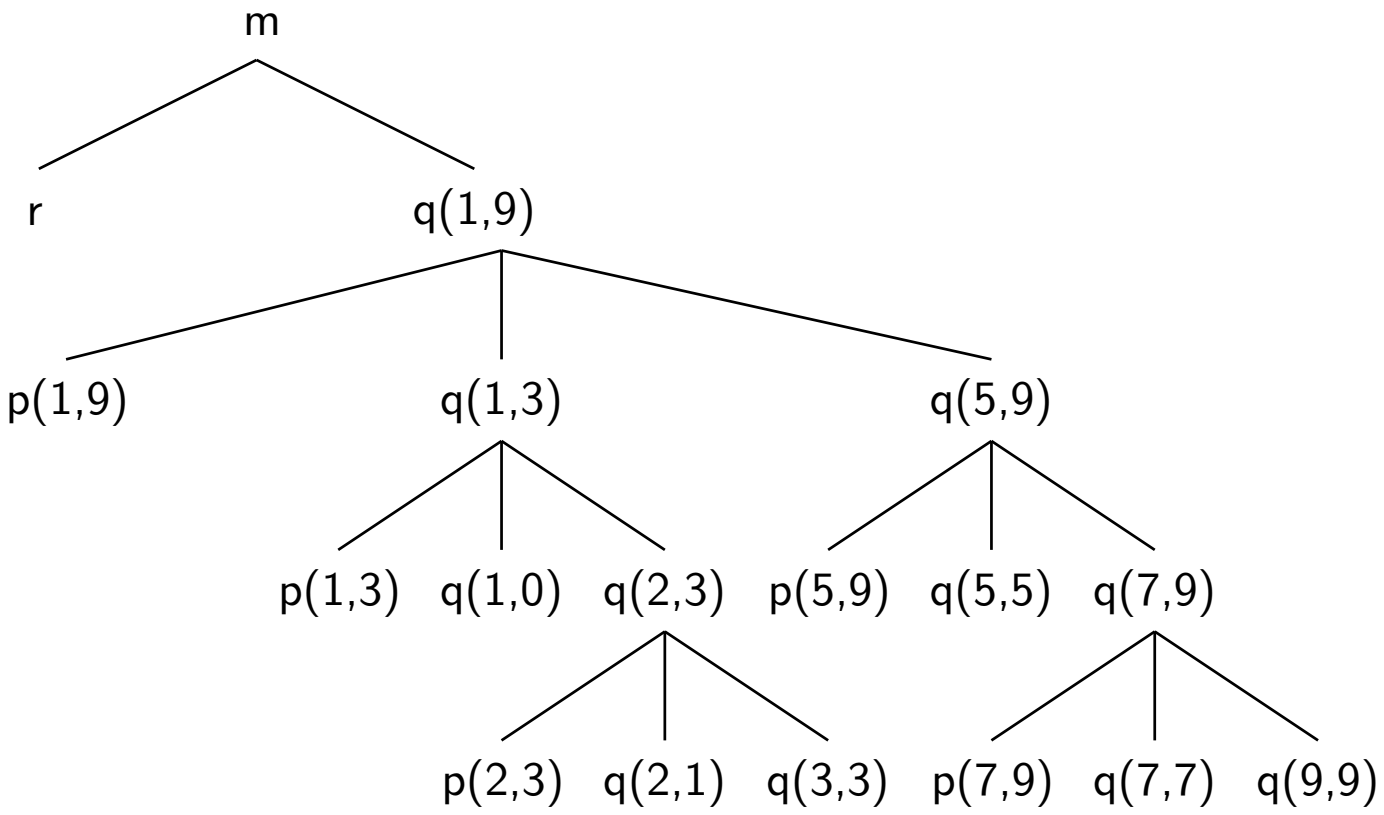
```
int array[11];
                              /* Partitionne array[m..n] de façon à ce que:    */
int partition(int m, int n)   /*   - array[m .. p-1] contient les valeurs < v  */
{ /* ... */ }                 /*   - array[p]        contient le pivot v       */
                              /*   - array[p+1 .. n] contient les valeurs >= v */
void quicksort(int m, int n)
{
  if (n > m)
    {
      int i = partition(m, n);
      quicksort(m, i - 1);
      quicksort(i + 1, n);
    }
}

main()
{
  readArray(array);  /* readArray retourne des valeurs quelconques  */
  array[0] = -9999;  /* Valeur plus petite que toute valeur à trier */
  array[10] = 9999;  /* Valeur plus grande que toute valeur à trier */
  quicksort(1, 9);
}
```

Arbres d'activation

Exemple 7.1 (suite).

Voici un arbre d'activation correspondant à une exécution de cet algorithme:



Piles de contrôle

Le flot de contrôle d'un programme correspond à une traversée en profondeur d'abord de son arbre d'activation.

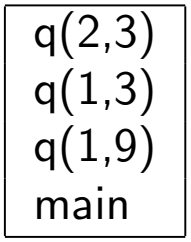
Grâce à cet ordre de traversée, on peut employer une *pile de contrôle* pour conserver les activations de procédures qui sont vivantes.

L'idée consiste à empiler le noeud d'une activation sur la pile de contrôle lorsque l'activation débute et à dépiler ce noeud lorsque l'activation se termine.

Lorsqu'un noeud n de l'arbre d'activation se trouve sur le dessus de la pile, alors le contenu complet de la pile consiste en un chemin de tous les noeuds de la racine jusqu'à n.

L'utilisation d'une pile de contrôle offre une certaine souplesse et permet d'accommoder de l'allocation dynamique d'espace de rangement en régime de pile.

Exemple: Si q(2,3) (de l'exemple précédent) est en cours d'exécution, la pile ressemble à ceci:



Portée des déclarations

Une déclaration est une construction syntaxique qui sert à associer une certaine information à un nom, que ce soit explicitement ou implicitement.

Or, il peut y avoir plusieurs déclarations distinctes du même nom dans un même programme.

Ce sont les *règles de portée* qui déterminent quelle déclaration est en cause pour une apparition donnée d'un nom.

La portion du programme où une certaine déclaration est en cause pour un nom donné s'appelle la *portée* de la déclaration.

Portée des déclarations

Une apparition d'un nom est *locale* lorsque l'apparition est dans la portée d'une déclaration qui est faite à l'intérieur d'une procédure. Autrement, l'apparition est *non-locale*.

Lorsque le contexte permet de lever l'ambiguïté, on se permet un abus de langage en parlant de la "portée d'un nom '`x`'" alors qu'on veut signifier la "portée de la déclaration du nom '`x`' qui est en cause pour cette apparition de '`x`'".

À la compilation, la table des symboles peut être utilisée pour retrouver la déclaration s'appliquant à une certaine apparition d'un nom. La présence d'une déclaration cause la création d'une entrée dans la table des symboles. Tant qu'on se trouve dans la portée de cette déclaration, c'est cette entrée qui est retournée lorsqu'on fait une recherche sur le nom en question.

Liaisons des noms

Exemple de portées

```
{
  int x, y;

  /* Section 1 */
  x = ...y...;

  {
    int y, z;

    /* Section 2 */
    y = ...z...x...;
  }

  /* Section 3 */
  y = ...x...
}
```

Déclaration	**Portée**
de x	sections 1, 2 et 3
de y (1ère)	sections 1 et 3
de y (2ème)	section 2
de z	section 2

Liaisons des noms

Un nom donné peut correspondre à plus d'un objet à l'exécution et ce, même si ce nom n'est déclaré qu'une seule fois.

L'*objet* dont il est question ici est l'espace de rangement capable de contenir des valeurs.

En sémantique des langages de programmation, le terme *environnement* désigne une fonction qui associe des noms à des emplacements de rangement et le terme *état* désigne une fonction qui associe des emplacements de rangement à des valeurs.

La *lecture* d'une variable nommée `x` consiste en:

1. la consultation de l'environnement pour connaître l'emplacement l où est stockée `x` et
2. la consultation de l'état pour connaître la valeur stockée en l'emplacement l.

L'*écriture* d'une valeur v dans une variable nommée `x` consiste en:

1. la consultation de l'environnement pour connaître l'emplacement l où est stockée `x` et
2. la modification de l'état à l'entrée l pour y inscrire v.

Liaisons des noms

Une affectation (par exemple, '`x := e;`') modifie l'état mais pas l'environnement.

Au contraire, "l'entrée en vigueur" d'une déclaration (par exemple, '`int x;`') modifie l'environnement mais ne modifie pas nécessairement l'état.

Si un environnement associe un nom '`x`' à un emplacement de rangement s, on dit que '`x`' est *liée* à s. L'association elle-même s'appelle la liaison de '`x`'.

Le terme "emplacement" est générique ici. Si le type de '`x`' est complexe, il peut y avoir plusieurs emplacements (espaces en mémoire ou registres) qui sont réservés pour contenir la valeur de '`x`'.

Notion statique	Équivalent dynamique
définition d'une procédure	activation de la procédure
déclaration d'un nom	liaison d'un nom
portée d'une déclaration	durée de vie d'une liaison

Liaisons des noms

Si on utilise la terminologie des l-*values* et des r-*values*, alors on dit qu'un environnement associe des noms à des l-values et qu'un état associe des l-values à des r-values.

Les termes l-value et r-value sont des contractions des termes "left-value" et "right-value", pour *valeur à gauche* et *valeur à droite*.

Les deux termes réfèrent aux deux interprétations que l'on peut donner aux expressions selon qu'elles sont placées à gauche ou à droite d'un opérateur d'affectation.

Par exemple:

- dans `t[i] := ...`, ce qui nous intéresse de `t[i]`, c'est sa l-value; il s'agit de l'espace de rangement, dénoté par `t[i]`, où doit se faire l'écriture;
- dans `... := t[i]`, ce qui nous intéresse de `t[i]`, c'est sa r-value; il s'agit de la valeur qui est lue depuis l'espace de rangement dénoté par `t[i]`.

Expression	l-**value**	r-**value**
`incr`	case en mémoire ou registre	valeur stockée dans la variable
`t[i]`	case en mémoire	valeur stockée dans la case
1234	—	entier valant 1234
`nomFonctionEnPascal`	case en mémoire	—

Comment gérer le rangement et les liaisons?

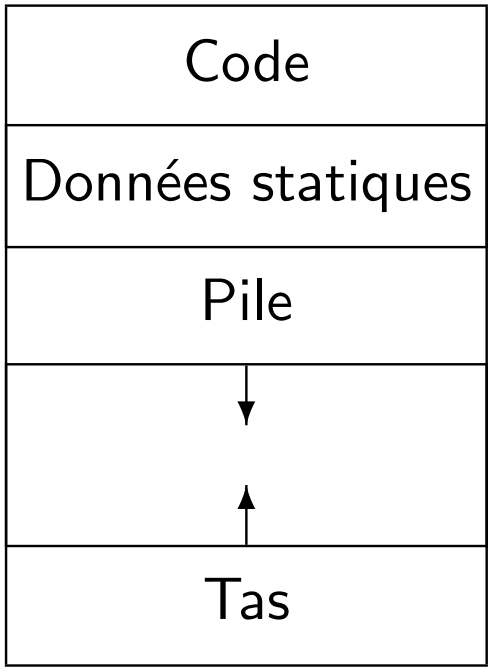
La méthode qui doit être employée pour gérer ces entités dépend de la réponse aux questions suivantes:

1. Est-ce que les procédures peuvent être récursives?
2. Qu'arrive-t-il à la valeur des noms locaux lorsque le contrôle termine l'activation d'une procédure?
3. Est-ce qu'une procédure peut référer aux noms non-locaux?
4. Comment les paramètres sont-ils passés lors d'un appel?
5. Les procédures peuvent-elles être passées comme arguments?
6. Les procédures peuvent-elles être retournées comme valeurs de retour?
7. Le programme a-t-il la capacité d'allouer de l'espace de rangement dynamiquement?
8. Est-ce que l'espace de rangement doit être libéré explicitement?

Organisation de l'espace de rangement

L'organisation de l'espace de rangement vue dans cette section est utilisable dans le cas de langages comme Fortran, Pascal, C et Java.

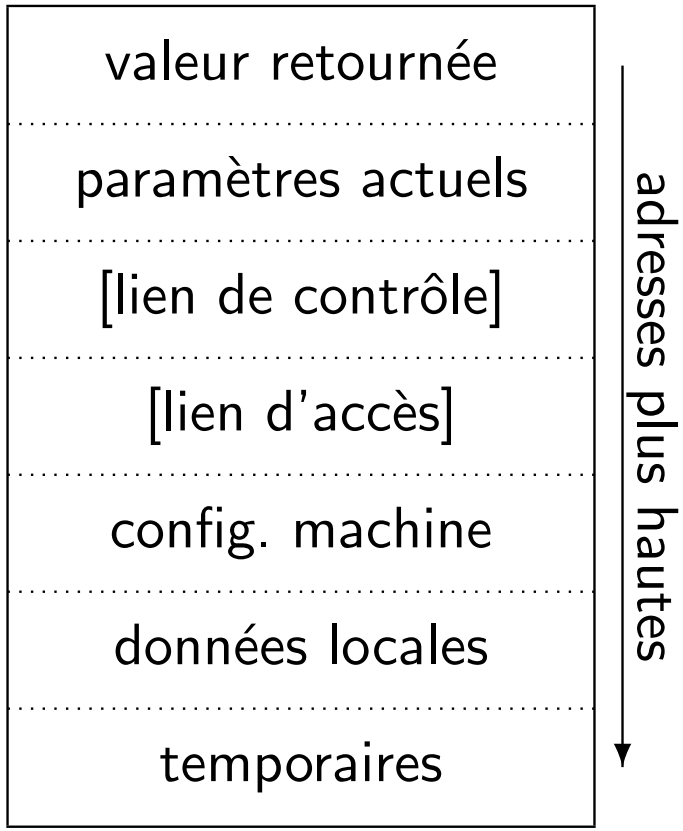
Séparation de l'espace mémoire alloué au programme à l'exécution:



Structures d'activation

L'information reliée à l'activation d'une procédure est stockée dans une *structure d'activation*.

Dans le cas le plus général, une structure d'activation comporte les champs suivants:



Infrastructure d'exécution:

Allocation de la mémoire

Stratégies d'allocation de l'espace mémoire

Les stratégies suivantes sont utilisées dans les trois zones de données d'un programme:

1. l'allocation statique détermine la disposition finale des données;
2. l'allocation par pile gère l'espace à la manière d'une pile;
3. l'allocation en tas permet l'allocation et la libération des objets sans contraintes sur l'ordre où ces opérations sont effectuées.

Allocation statique

L'allocation statique permet de lier chaque nom à un emplacement fixe et ce, dès la compilation.

L'allocation statique est couramment utilisée pour les variables globales mais permet aussi à des noms locaux de devenir *persistents*; c'est-à-dire que la valeur de ces variables est conservée d'une activation d'une procédure à l'autre.

Avantages et inconvénients:

+ Allocation la plus simple qui soit.
+ Allocation la plus efficace qui soit: plus rien à faire à l'exécution.
– Il n'y a pas d'allocation dynamique possible.
– La taille et les contraintes d'alignement des objets doivent être connues à la compilation.
– Des procédures récursives générales ne peuvent pas être implantées.
– Les blocs d'activations et les données des procédures occupent de l'espace, même en l'absence d'activation.

Allocation par pile

L'allocation par pile profite du fait qu'une pile de contrôle est déjà utilisée pour effectuer l'exécution du programme.

À chaque activation de procédure, on réserve (empile) un bloc de mémoire pour accommoder la structure d'activation et les variables locales à la procédure.

À la fin de l'activation, le bloc est libéré (dépilé). Conséquemment, l'espace de rangement qui servait à contenir la valeur des variables locales est libéré.

Puisque la position de la structure d'activation n'est pas nécessairement au même emplacement en mémoire d'une activation à l'autre, on ne peut connaître statiquement l'emplacement des variables.

Toutefois, on considère qu'un pointeur, appelé *top*, pointe constamment sur le dessus de la pile.

L'emplacement des variables devient alors calculable *en fonction de* top.

Allocation par pile

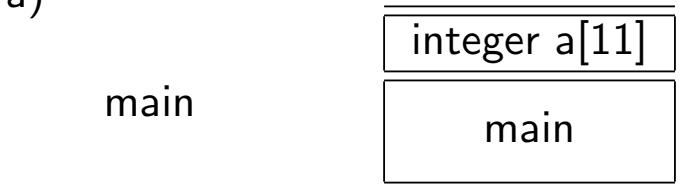
Avantages et inconvénients:

+ Allocation dynamique possible.
+ Permet la récursion générale.
+ Seuls les blocs d'activations et les données des procédures activées occupent de l'espace.
+ Allocation et libération rapides.
+ Accès aux données presqu'aussi efficace qu'avec l'allocation statique.
– Le compilateur doit pouvoir calculer statiquement la position relative des variables et des champs de la structure d'activation.
– Allocation dynamique... mais seulement en régime de pile.

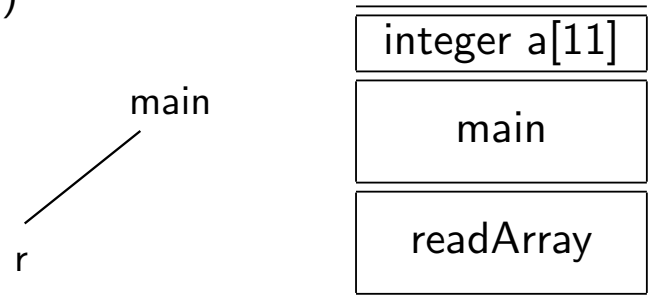
Allocation par pile

Exemple 7.4

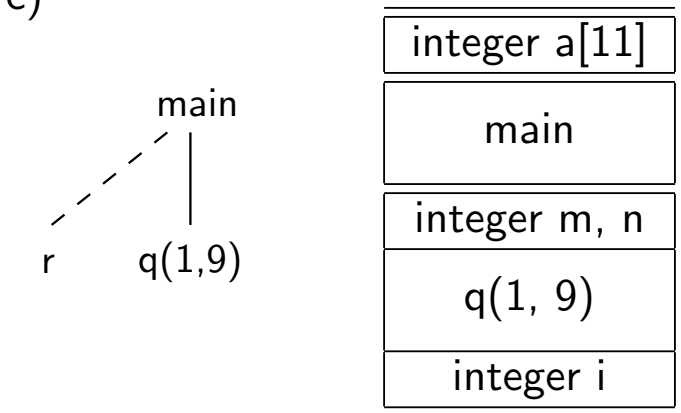
a)



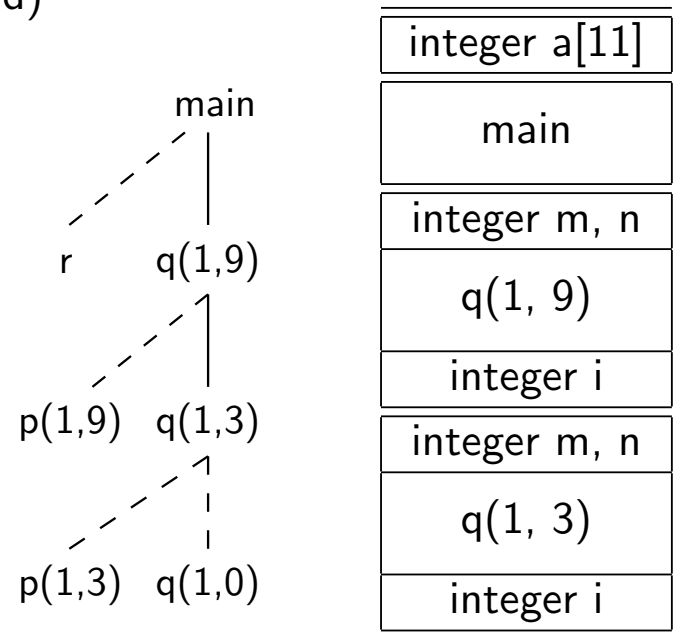
b)



c)



d)



Note: le pointillé indique un appel de fonction complété

Allocation par pile: séquences d'appel

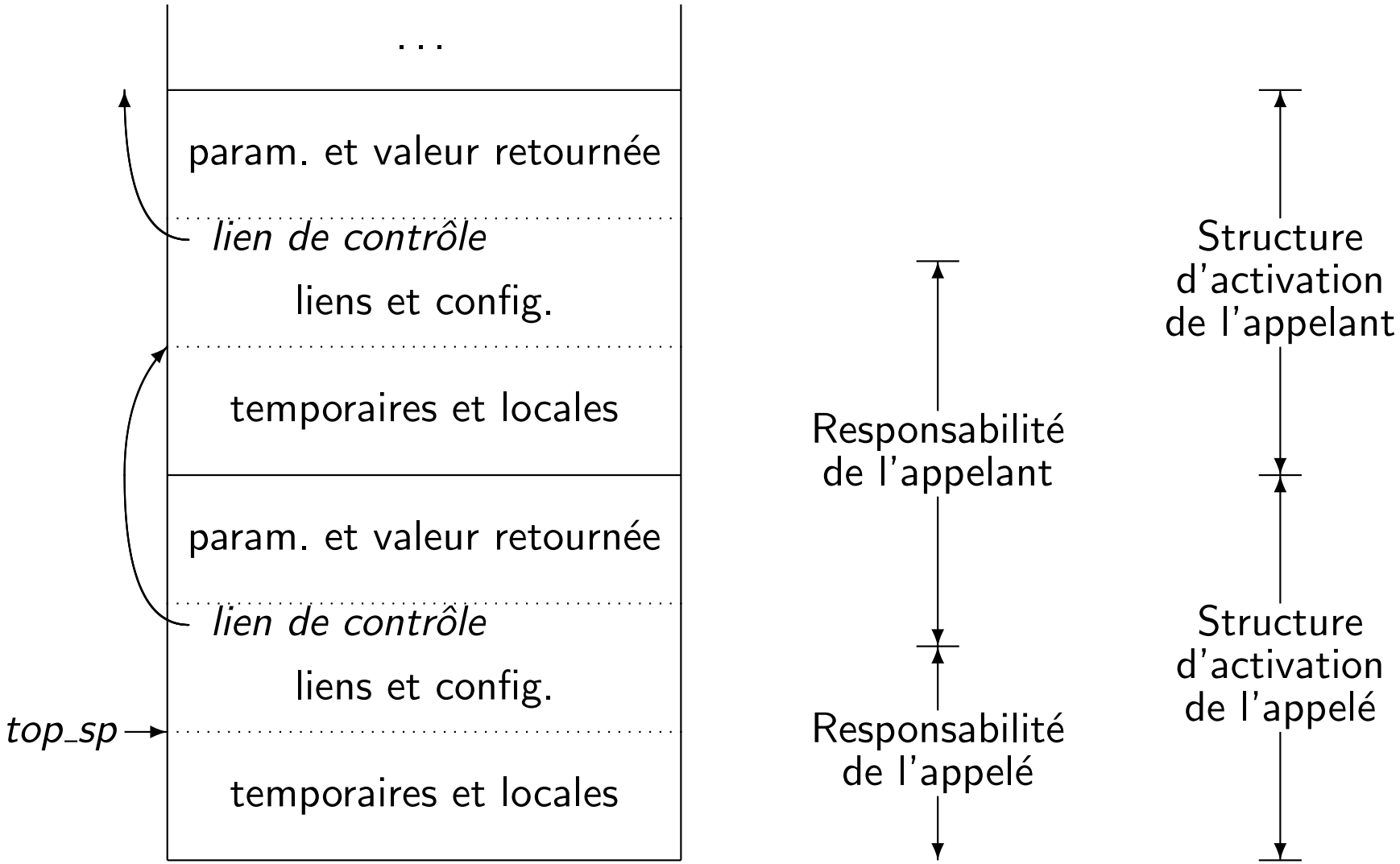
La mécanique des appels de fonctions est implantée à l'aide de *séquences d'appel* et de *séquences de retour*.

La séquence d'appel alloue la structure d'activation et initialise ses champs.

La séquence de retour installe la valeur de retour à l'endroit prévu à cet effet et libère la structure d'activation.

L'implantation de chacune de ces deux séquences est placée en partie chez l'*appelant* et en partie chez l'*appelé*. La partie des séquences dévolue à chacun de ces acteurs varie d'un langage à l'autre.

Allocation par pile: séquences d'appel



Allocation par pile: séquences d'appel et de retour

En prenant pour acquis qu'un pointeur supplémentaire, *top_sp*, pointe à un emplacement précis de la structure d'activation courante, on peut plausiblement décrire la séquence d'appel suivante:

1. L'appelant évalue les paramètres actuels.
2. L'appelant sauvegarde l'adresse de retour et l'ancienne valeur de *top_sp* et incrémente ensuite *top_sp*.
3. L'appelé sauvegarde la configuration de la machine.
4. L'appelé initialise ses données locales.

et la séquence de retour suivante:

1. L'appelé installe la valeur de retour à l'emplacement prévu près de la structure d'activation de l'appelant.
2. L'appelé restaure la configuration de la machine et le pointeur *top_sp* et branche ensuite à l'adresse de retour.
3. L'appelant récupère la valeur de retour et libère la structure d'activation de l'appelé.

Allocation par pile

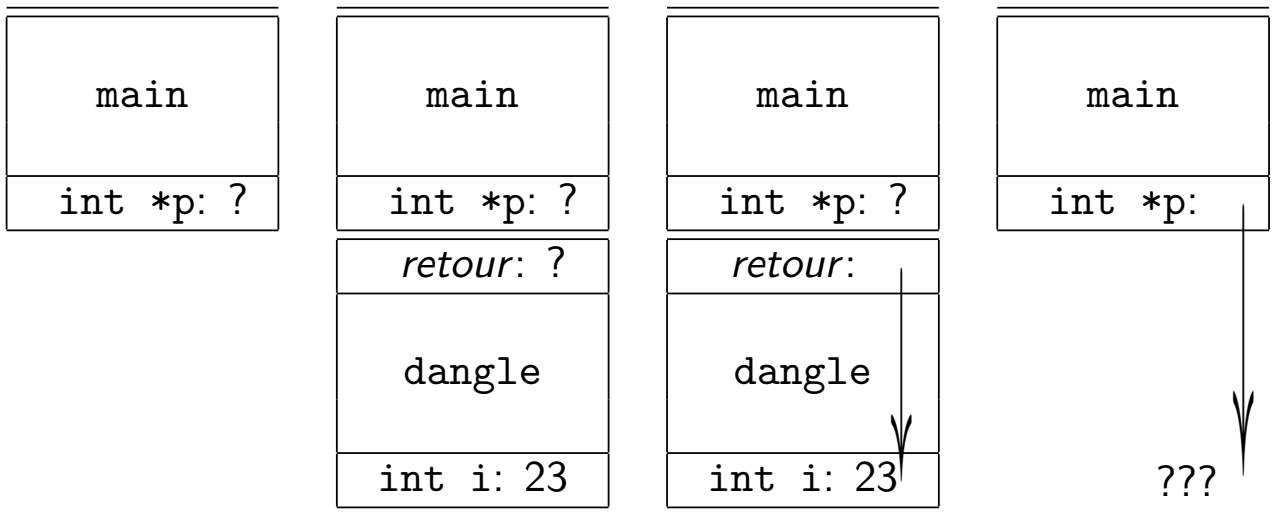
Grâce au pointeur supplémentaire *top_sp*, il est possible d'effectuer l'allocation sur la pile de données à taille variable (voir section **7.2.4**).

Un des effets de l'allocation par pile est de permettre d'obtenir l'adresse de variables qui n'existent plus. Plus précisément, c'est un effet de l'allocation dynamique combinée à la manipulation directe de pointeurs.

```
main()
{
  int *p;
  p = dangle();
  /* suite... */
}

int *dangle()
{
  int i = 23;
  return &i;
}
```

1. Obtention de l'adresse A grâce à `&i`: OK
2. Retour de A: OK
3. Usage de A dans "`suite...`": ILLÉGAL



Allocation en tas

L'allocation en tas devient nécessaire dans les situations suivantes:

- La valeur de certaines variables locales doit continuer à exister après l'activation d'une procédure. Exemple: fonctions de première classe.
- L'activation de l'appelé survit plus longtemps que l'activation de l'appelant. Exemple: optimisation de la récursion terminale.

Avantages et inconvénients:

+ Allocation la plus souple.
+ Possède la plupart des avantages de l'allocation par pile.
− L'allocation et la libération sont plus ou moins coûteuses.

La gestion d'un tas est un sujet très vaste où il n'y a pas une meilleure solution générale, particulièrement lorsque la libération des objets est automatique. (Voir sections 7.4 à 7.8 dans le manuel.)

Vulnérabilités dans les langages de programmation

Deux vulnérabilités typiques.

- Lectures dans des tampons sans contrôle de longueur;
  par exemple: `char *gets(char *s);`
  où, même si `s` pointe sur un tampon de taille n,
  plus de n caractères peuvent être lus,
  écrasant les données adjacentes au tampon (bloc d'activation).

- Lectures structurées avec un format sous le contrôle (total ou partiel) de l'environnement;
  par exemple: `int scanf(const char *format, ...);`
  où, si le `format` dépend des entrées venant de l'environnement,
  des commandes de lectures (%d, %c, etc.) imprévues peuvent y être ajoutées,
  résultant en des écritures imprévues en mémoire.