Eléments d'Informatique

Cours 8 – Sens d'un programme, pile d'exécution, compilation

Catherine Recanati

UNIVERSITÉ PARIS 13 NORD

Plan général

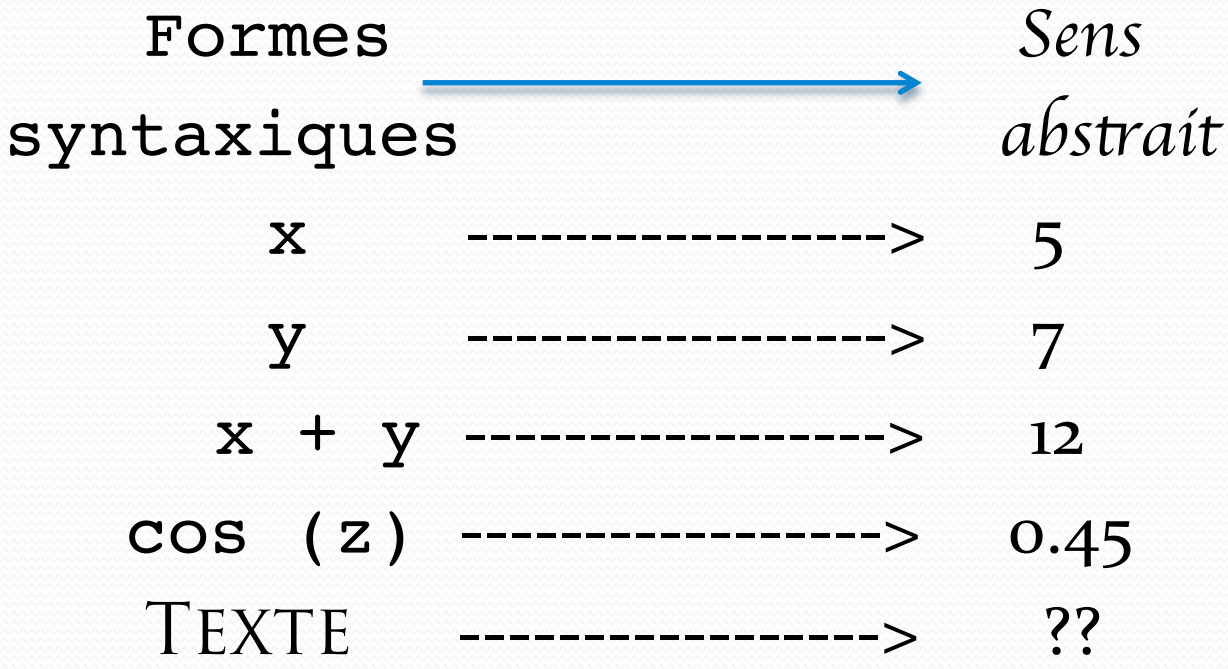
- **Représentation des nombres. Notion de variable.**
- **Programme. Expressions.**
- **Architecture des ordinateurs: langage machine, langage assembleur, AMIL.**
- **Systèmes d'exploitation : fichiers, processus, compilation.**
- **Instructions de contrôle: boucles et branchements.**
- **Programme. Définition de fonction. Appel fonctionnel.**
- **Tableaux de variables et fonctions d'arguments de type tableau.**
- **Sens d'un programme, pile d'exécution, compilation.**
- Pointeurs et tableaux.
- Chaines de caractères, bibliothèque <string.h>.
- Allocation dynamique, liste chaînées.
- Révisions.

- Cours 8 – Sens d'un programme, pile d'exécution, compilation

- Sens d'un programme
- Portée des variables
- Appel de fonction
- Organisation du code, pile d'exécution
- Compilation séparée, édition de liens, bibliothèques

Plan

Sens d'un programme : Interprétation sémantique



Plan

Sens d'un programme : Interprétation sémantique

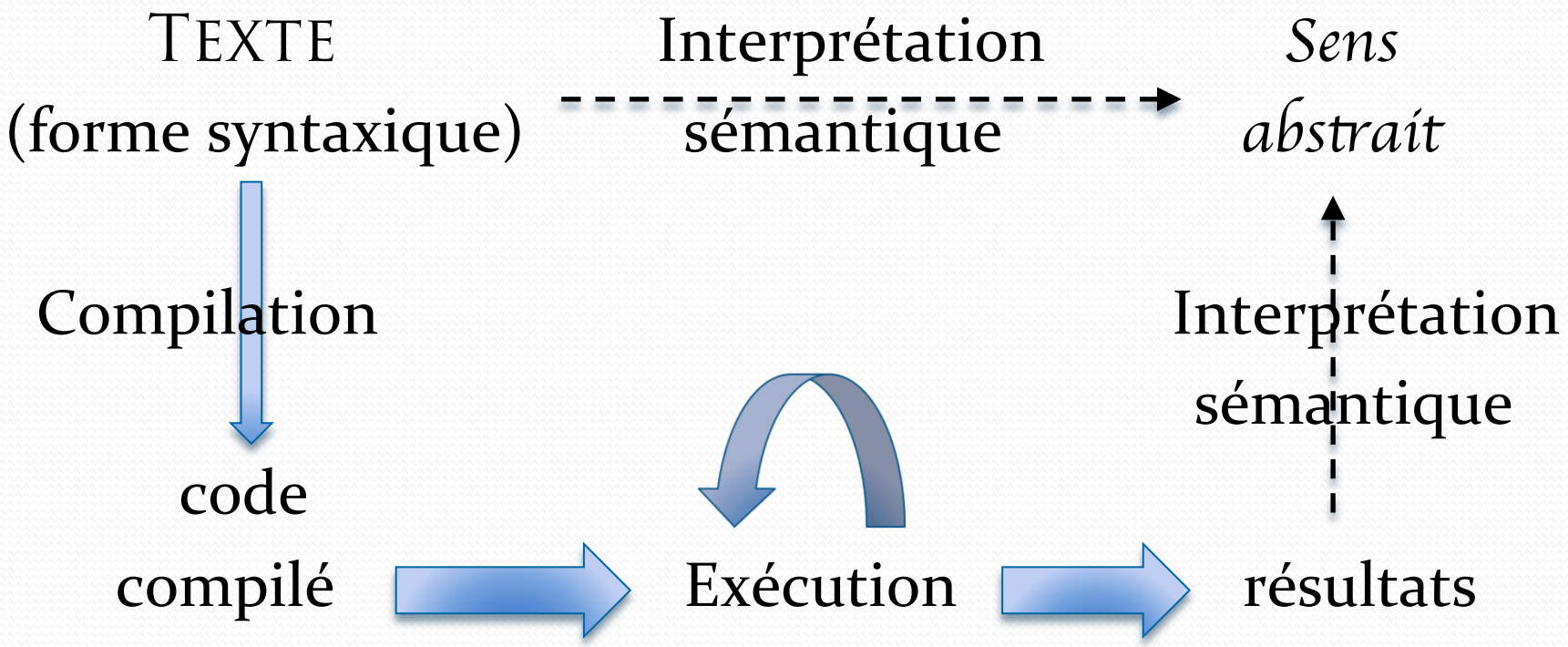
Plus précisément, à tout couple (forme syntaxique, environnement) on peut associer un sens (une valeur).

Φ:(`forme synt.`,Env) ⟶ *valeur*

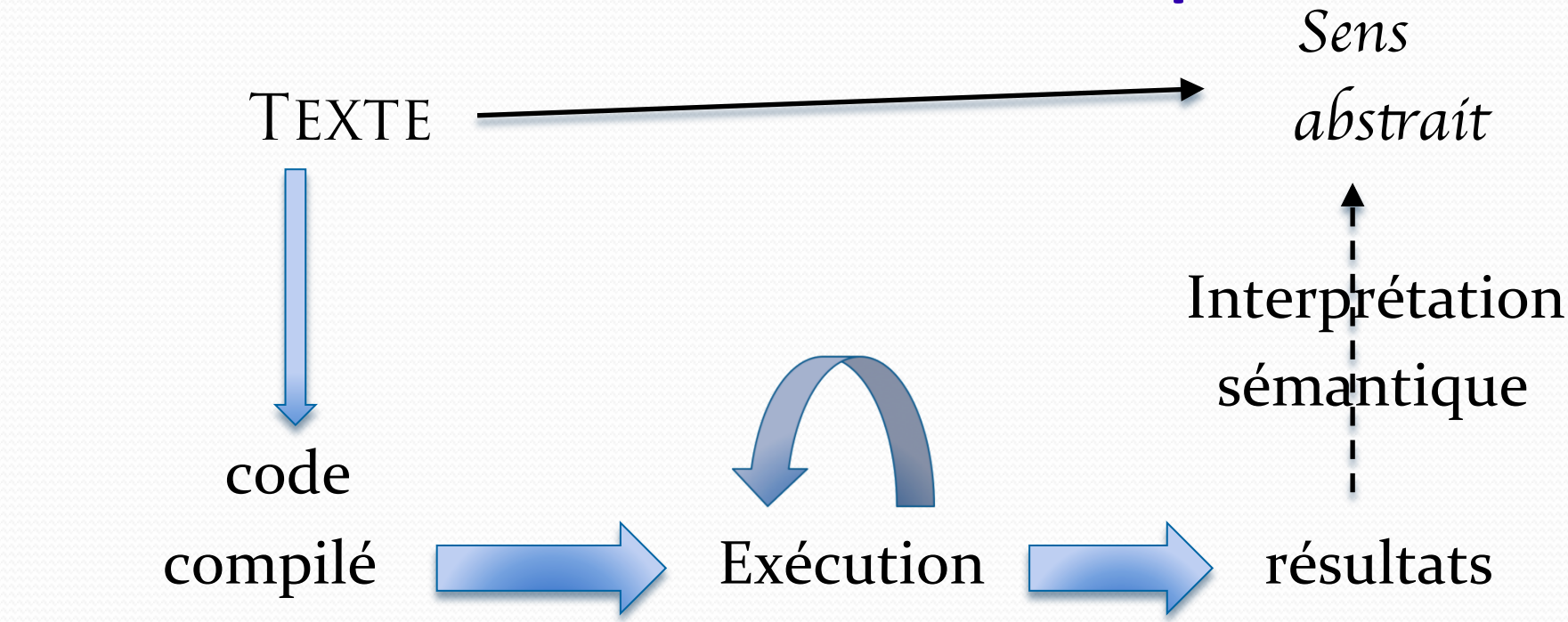
`(x + y, <x,5>` ⟶ 12
`<y,7> )`

Un environnement est un ensemble de liaisons <nom, valeur>.

Sens d'un programme : Interprétation sémantique



Correction sémantique



Exec: données1 ---------- result1
données2 ---------- result2
données3 ---------- result3

Correction sémantique

```
main () {
   int x ;  /* x codé sur 32 bits */
   printf("Entrez x = ");
   scanf("\n%o", &x);
   printf("2 fois %d = %d\n",x,2*x);
}
```

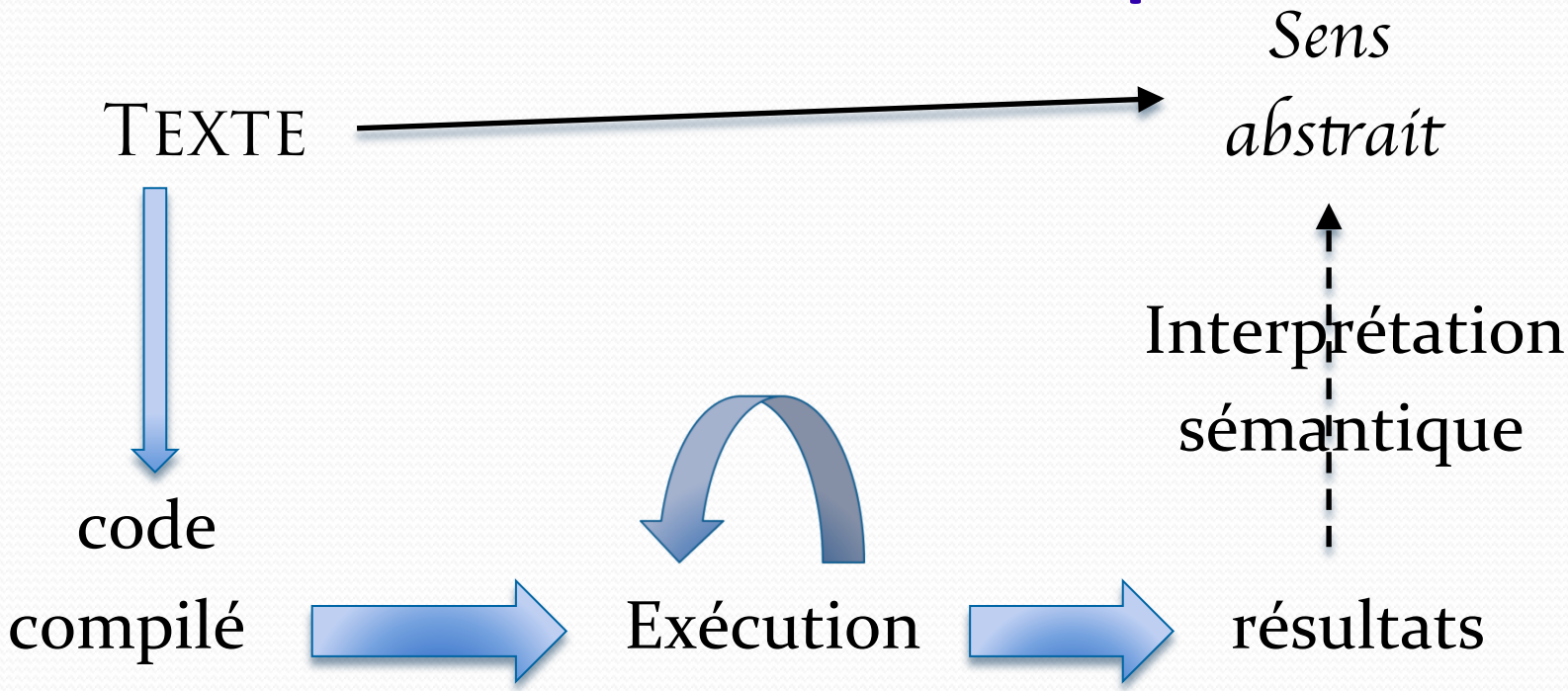
- `Entrez x = 1111111111` (30 bits significatifs)

`2 fois 153391689 = 306783378`

- `Entrez x = 10000000000` (31 bits significatifs)

`2 fois 1073741824 = -2147483648`

Correction sémantique



Exec: `o777` ----------> correct

`o1111111111` ----------> correct

`o10000000000` ----------> `incorrect`

Plan

Sens et compilation

Le compilateur doit effectuer une traduction du texte source du programme, en un code machine interprétable. Cette traduction n'est possible que si le texte source est syntaxiquement correct.

Pour être sémantiquement correcte (du point de vue du compilateur), cette traduction ne doit pas signaler de *warning*.

Plan

Sens et compilation

Mais pour être réellement validée, la modélisation souhaitée par le programmeur devra être précisée (envisager un jeu de données pour tester le programme) et il faudrait idéalement faire une « preuve » de ce que le programme fait.

ex: calcul du double d'un entier (type int codé sur 32 bits). Pb du débordement.

Plan

Sens et compilation

Quoiqu'il en soit, le langage C précise bien les règles d'interprétation des différents énoncés d'un programme et le compilateur s'engage à les respecter.

Ainsi, la déclaration de variables (ou de fonctions) permet d'énoncer des règles de portées. Et l'appel fonctionnel permet d'introduire la notion de cadre (ou environnement) de calcul.

Plan

Portée lexicale des variables

Pour trouver la déclaration (qui déterminera une attribution d'adresse) d'une variable x dans un bloc, on recherche d'abord sa déclaration au niveau du bloc. Si elle n'y est pas, on la cherche au niveau du bloc du dessus (bloc englobant), et ainsi de suite. ...

```
main {
    ... {Bloc B
        ... {Bloc A: x ? }
```

Plan

Portée lexicale

```
int main() {
    int x = 2;
    int y;

    y = x;
    {int x=0;
        printf("x du bloc =%d ", x);
    }
    printf("x du main =%d ", x);
    return EXIT_SUCCESS;
}
```

Plan

Portée lexicale

```
int main() {
     int x = 2;
     int y;

     y = x;
     for (int x=0; x<y  ;x++) {
          printf("x du bloc =%d ", x);
     }
     printf("x du main =%d ", x);
     return EXIT_SUCCESS;
}
```

```
x du bloc =0 x du bloc =1 x du main =2
```

Plan

Portée lexicale

```
int f(int x) {
   …
}

int main() {
    int x = 2;

    x = f(x - 1);
    printf("x vaut %d", x);
    return EXIT_SUCCESS;
}
```

Plan


```
int f(int x) {
     if (x <= 0)
          return -1;
     else
          return  f(x – 1);
}

int main() {
     int x = 2;

     x = f(x - 1);
     printf("x vaut %d", x);
     return EXIT_SUCCESS;
}
```

Plan


```
int f(int x) {
    if (x <= 0)
        return -1;
    else
        return  f(x – 1);
}

int main() {
    int x = 2;

    x = f(x - 1);
    printf("x vaut %d", x);
    return EXIT_SUCCESS;
}
```

Plan


```
int f(int x) {
    if (x <= 0)
        return -1;
    else
        return  f(x – 1);
}

int main() {
    int x = 2;
    x = f(x - 1);
    printf("x vaut %d", x);
    return EXIT_SUCCESS;
}
```

Affichera "x vaut -1"

return -1

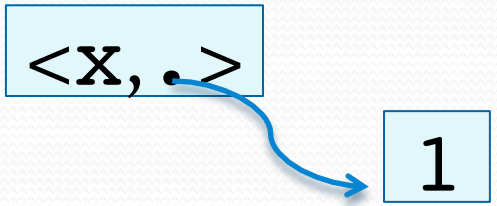
f appelée avec argument de valeur 0

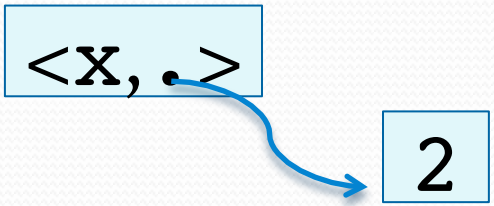
f appelée avec argument de valeur 1

Changement d'environnement

```
int f(int x) {
    if (x <= 0)
        return -1;
    else
        return  f(x – 1);
}

int main() {
    int x = 2;
    x = f(x - 1);
```

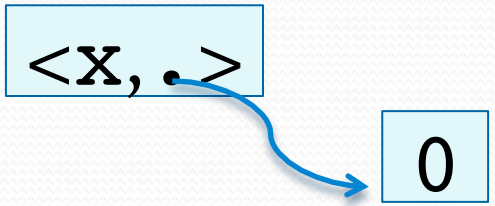


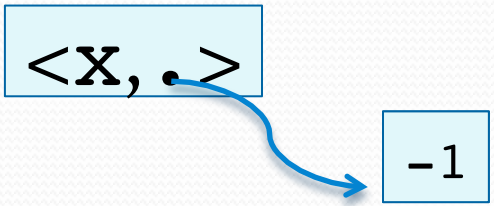


Changement d'environnement

```
int f(int x) {
    if (x <= 0)
        return -1;
    else
        return  f(x – 1);
}

int main() {
    int x = 2;
    x = f(x - 1);
```





Plan

Organisation du code

Le compilateur doit assurer les changements d'environnements entre les différents blocs - en respectant les règles de portée de déclaration de variables, et celles de liaison des paramètres en cas d'exécution d'un appel de fonction.

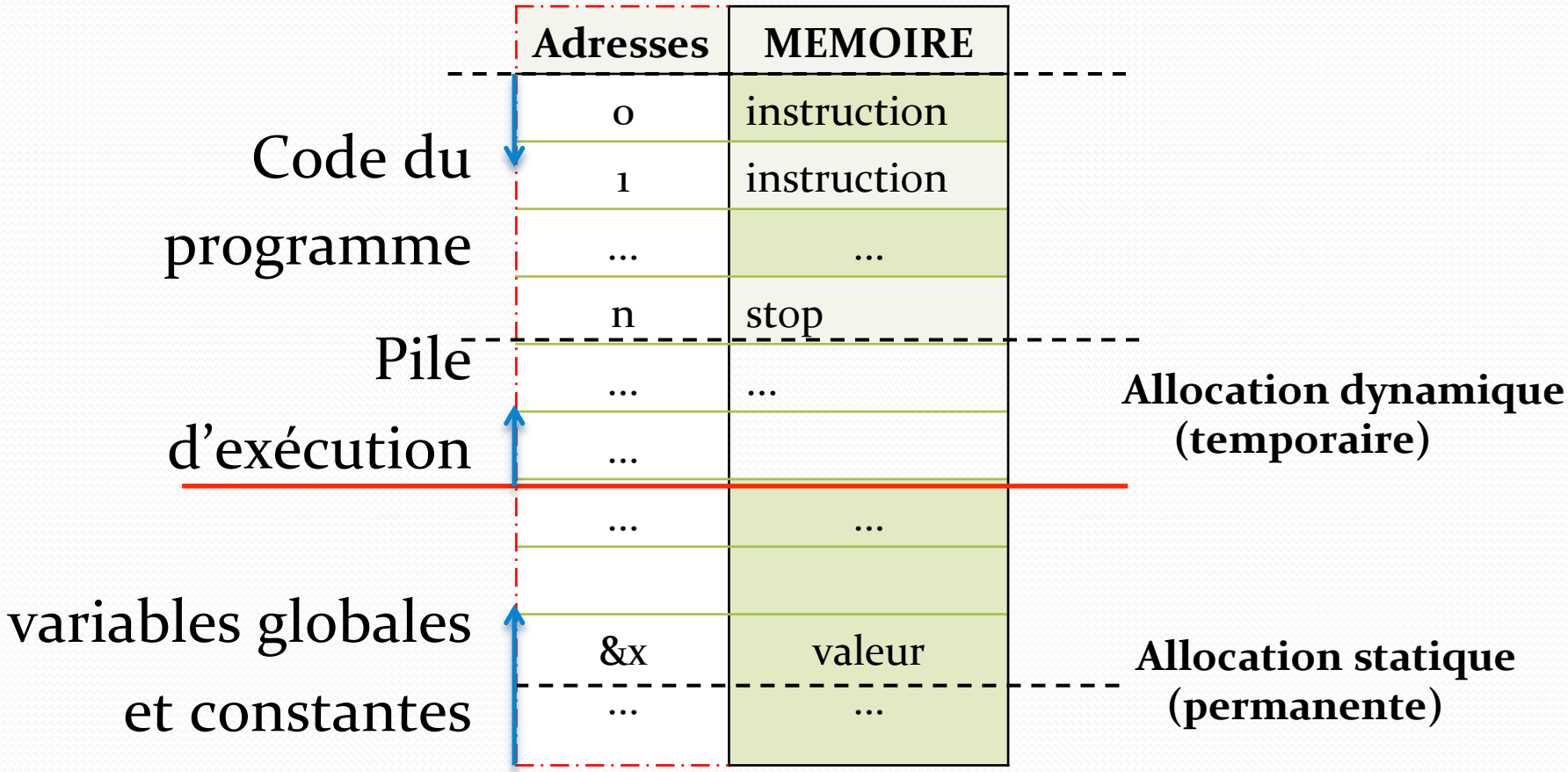
Plan

Organisation du code

Pour respecter la sémantique du langage, le compilateur gère dynamiquement les allocations mémoire des différentes variables (paramètres de fonction, variables locales à un bloc d'instructions ou à une fonction).

L'organisation de l'exécution du code est basée sur l'empilement (et le dépilement) de cadres constitués de liaisons <variable, adresse> et d'adresse de retour. Cet empilement de cadres s'appelle la pile d'exécution (ou pile d'appels).

Organisation du code : Segmentation de la mémoire



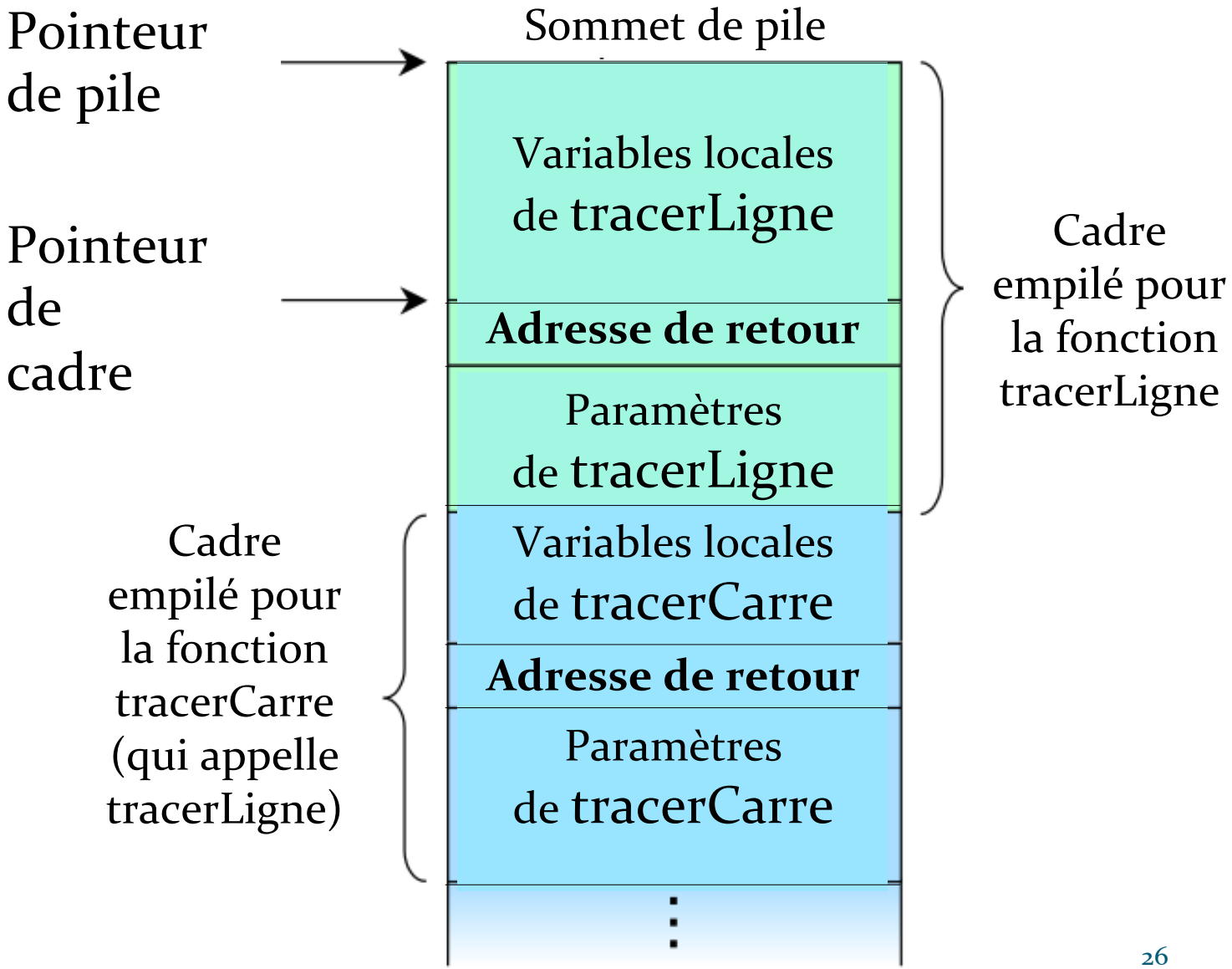
Plan

Pile d'appels de fonctions

Un calcul d'appel de fonction s'effectue:

- en liant aux paramètres de la fonction la valeur des expressions en position d'arguments dans l'expression d'appel (le calcul est effectué dans l'env. d'appel). On empile alors ces liaisons, et l'adresse de retour du calcul.
- en exécutant ensuite le corps de la fonction dans ce nouvel environnement jusqu'au return. On dépile alors cet env. et on reprend le calcul précédent.

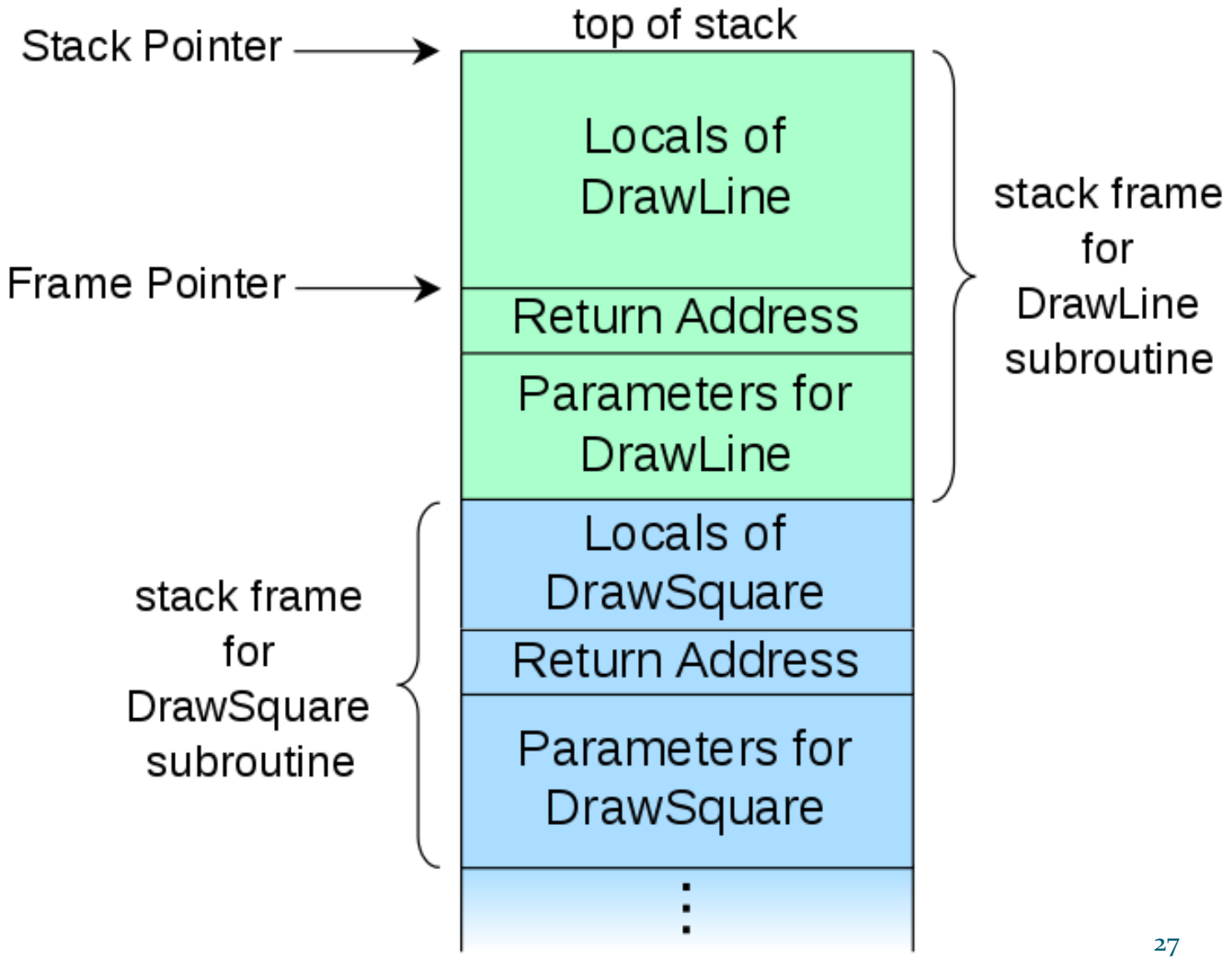
Exemple: tracerCarre appelle tracerLigne



Plan

- Sens d'un programme
- Portée (lexicale) des variables
- Appel de fonction
- Organisation du code
- Pile d'exécution
- Compilation séparée
- Editeur de liens, bibliothèques

En anglais ...



Plan

Pile d'exécution

Remarque: Si le programme est interrompu par l'impression d'une erreur d'exécution, on verra l'affichage des noms de fonctions figurant dans la pile d'exécution au moment où est survenu l'erreur.

Les debugueurs permettent aussi d'afficher la pile d'exécution.

Plan

Compilation séparée

On a vu qu'on pouvait donner au compilateur des fichiers contenant juste des déclarations de variables et des définitions de fonctions. La commande

```
$ gcc –c fichier.c
```

crée un code objet dans fichier.o, mais ne crée pas de fichier exécutable a.out. Le fichier créé, fichier.o, n'est pas directement exécutable mais il pourra être relié à un code exécutable plus tard.

Compilation séparée

On peut compiler séparément un fichier contenant la définition d'une fonction fonct et intitulé fonct.c :

```
$  gcc –c fonct.c
```

on obtient un fichier fonct.o qui pourra ensuite être combiné avec un fichier main.c pour créer un exécutable avec :

```
$  gcc  fonct.o  main.c
```

ou

```
$  gcc  fonct.o  main.c –o exe
```

Compilation séparée

On peut aussi créer un exécutable à partir de plusieurs fichiers, certains compilés, d'autres pas, grâce à gcc :

```
$ gcc declare.h file1.o file2.o file3.c main.c
```

La commande `gcc` est une commande complexe, qui lance, en fonction de ses arguments, d'autres commandes en arrière-plan (*background*). En particulier la commande `as` (pour assembleur), et la commande `ln` (pour *linker*).

Editeur de liens

La commande `as` (assembleur) assemble des morceaux de codes (avec des adresses inconnues) et produit un code objet (fichier .o). La commande `ln` (éditeur de liens) effectue les liens entre des adresses encore inconnues et le reste du code, pour produire finalement un code exécutable (fichier .exe sous Windows, a.out ou un nom quelconque sous Linux).

Plan

Editeur de liens

C'est la commande `ln` qui est chargée d'incorporer le code objet des fonctions d'une bibliothèque (*library*). Elle a besoin de connaître les adresses où se trouve le code objets des fonctions de la bibliothèque. C'est pourquoi la commande `gcc` (qui appelle `ln`) accepte des options indiquant où trouver ces codes de fonctions de librairies (cf. –l et –L).

Sans indication, ces codes objets sont cherchés en un emplacement standard de l'arborescence des fichiers (/usr/lib).

Plan

Bibliothèques

Nous verrons en TD/TP comment créer notre propre bibliothèque de fonctions à partir de codes objets archivés.

De même que l'utilisation de la librairie standard nécessite d'inclure le fichier de déclarations des fonctions qui y figurent (stdlib.h), on prévoiera de faire un fichier de déclarations à inclure dans tout programme souhaitant utiliser notre librairie.

Plan

Merci pour votre attention !

Des questions ?