ESIAL – TELECOM NANCY 2012 – 2013

Cours de Traduction I

Deuxième année

Partie n°2 : chapitre 4 et première partie du chapitre 5.

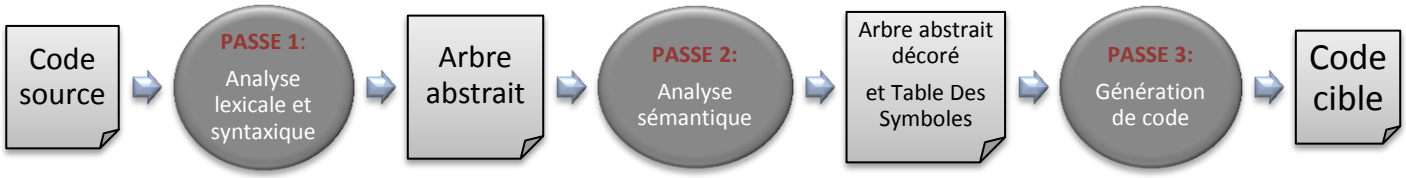
Table des matières

Chapitre n°4 : Arbres abstraits et contrôles sémantiques

Principes généraux

Rappels et compléments sur le compilateur

En compilation, une **passe** correspond à une lecture des données source afin de les transformer :



- **PASSE N°1 :**

La première lecture est celle du texte source qui est lu par l'analyseur lexical et l'analyseur syntaxique ; ceux-ci travaillent de pair. **L'arbre abstrait** est construit pendant l'analyse syntaxique.

- **PASSE N°2 :**

La seconde lecture des données (arbre généré à la fin de la première passe) a lieu durant l'analyse sémantique, qui intervient une fois l'analyse syntaxique terminée. **L'arbre** généré à la fin de la passe n°1 est **décoré** (on ajoute des informations aux éléments de cet arbre), et la **table des symboles (TDS)** est créée et remplie.

- **PASSE N°3 :**

Le générateur de code parcourt les données, c'est-à-dire maintenant l'arbre abstrait décoré et la TDS, afin de produire le **code cible**.

L'analyseur sémantique effectue un ensemble de contrôles sémantiques : il vérifie si le programme est correct et s'il un sens, en se référant au manuel de références fourni avec la syntaxe du langage source, comme par exemple :

- ✓ Toute variable est déclarée avant d'être utilisée (avant d'être lue ou affectée).
- ✓ Le programme ne contient pas de double déclaration, c'est-à-dire qu'un même nom de variable n'est pas utilisé pour deux déclarations différentes (sauf si ces déclarations ont lieu dans des blocs différents).
- ✓ Les types sont cohérents lors des opérations (par exemple, on n'affecte pas une chaîne de caractères à une variable déclarée comme étant de type entier).
- ✓ Les paramètres passés à une fonction (lors de son utilisation) sont de même type que les paramètres définis lors de la déclaration de la fonction.

Définitions

Espace de noms :

Ensemble de ce qui est désignable dans un contexte donné par une méthode d'accès faisant usage de noms symboliques. Autrement dit, cela désigne un lieu abstrait (méthode, bloc, fichier…) conçu pour accueillir un ensemble de termes (variables ayant un nom défini).

Exemple :

```
void f1(){
        int i=2;
        …
}
void f2(){
        float i = 2.5;
        …
}
```

Les intérieurs des fonctions **`f1()`** et **`f2()`** peuvent être vus comme deux espaces de noms différents. Une variable (ici, comme **i**) est unique dans son espace de noms. Dans la suite du cours, on parlera de **région** ou de **bloc**.

Région ou bloc :

Une région dans un programme informatique correspond à l'intérieur d'une fonction, ou d'une structure, ou encore d'un fichier. Si on reprend l'exemple en **C** précédent, on peut dire que l'intérieur de la fonction **fonctionA** est une région (ou un bloc). Le contenu complet du fichier est également une région. Ainsi, la région de la fonction **fonctionA** se trouve *dans* la région du fichier **C** : la première est donc **imbriquée** dans la seconde. Certains langages (comme le **Pascal**) autorisent la création de fonctions à l'intérieur d'une autre fonction. Par conséquent, le nombre d'imbrications dans un programme peut devenir assez important.

Règle de portée des déclarations :

Décrit la partie du programme (par exemple, un espace de noms, ou une région) dans laquelle les déclarations (de variables, ou de fonctions) sont valides. Il peut arriver qu'une variable déclarée localement dans un bloc ne soit accessible que dans ce bloc. Cependant, quand un premier bloc est imbriqué dans un second, alors le premier a généralement le droit d'accéder aux variables déclarées dans le second. Ce sont donc les règles de portée qui déterminent ce qui est autorisé et ce qui ne l'est pas.

Règle de visibilité :

Détermine dans quelle(s) partie(s) du programme un identifiant est visible.

Les règles permettent d'établir qu'à chaque identifiant correspond une déclaration unique. Ces règles vont donc permettre de faire **l'identification** qui consiste à associer à chaque identifiant utilisé sa déclaration. L'identification servira aux contrôles sémantiques statiques.

Exemple en C:

```
int i=2;   //Variable globale

void f(){
       int a = i+1 ;
       …
}
```

En **C**, la variable globale **i** est visible dans la fonction **f().**

Pile des régions ouvertes :

La pile des régions ouvertes est une pile (d'entiers) utilisée lors de l'analyse sémantique (ou syntaxique, selon la méthode utilisée, voir plus bas). On associe à chaque bloc (ou région) dans le programme un numéro entier unique. En parcourant l'arbre abstrait, dès qu'une région s'ouvre (par exemple, une accolade ouvrante), alors on empile son numéro dans cette pile. Dès que la région est refermée (accolade fermante), le sommet de la pile est dépilé. Cela permet de déterminer d'une part le nombre d'imbrications maximal dans le programme ; d'autre part, on connaît le nombre d'imbrications de chaque région du programme.

L'arbre abstrait (AST)

1) Définition

L'analyse syntaxique nous fournit un **arbre syntaxique « pur »**, appelé aussi **arbre de dérivation**. Celui-ci est obtenu directement d'après les règles de la grammaire et l'analyse du code source. Il contient cependant de nombreux nœuds superflus, voire inutiles pour l'analyse sémantique. Ainsi, pour faciliter le traitement de l'analyse sémantique, on va **simplifier et transformer cet arbre de dérivation**, pour en faire un **arbre syntaxique abstrait** (*Abstract Syntaxic Tree*, ***AST*** en anglais). Cette simplification **a lieu pendant l'analyse syntaxique**, où l'on remplace et modifie en temps réel les branches de l'arbre en suivant un certain nombre de règles de transformation (appelées **actions sémantiques**).

Ainsi, l'arbre abstrait constitue une **interface plus naturelle** entre l'analyse syntaxique et l'analyse sémantique.

Dans la suite du cours, on appellera simplement un *arbre syntaxique abstrait* un ***arbre abstrait*** (ou **AST**).

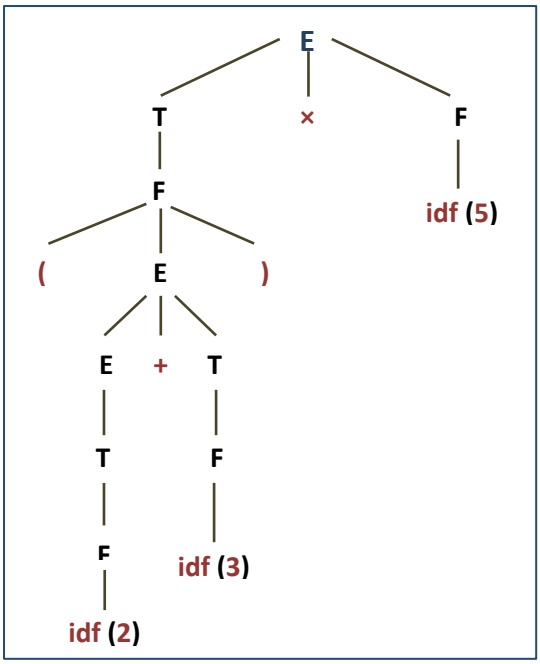
2) Exemple : transformation de l'arbre de dérivation en un arbre abstrait.

Grammaire:

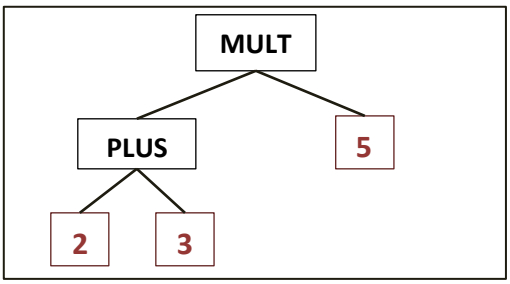
E → E+T | E-T | T
T → T×F | T/F | F
F → (E) | idf | -E

Texte source:

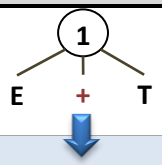
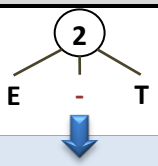
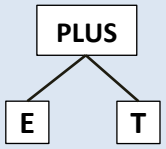
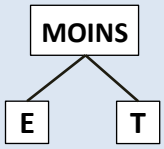
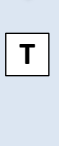
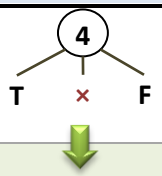
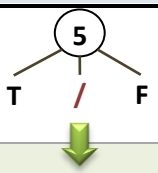
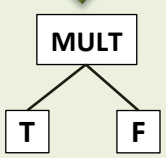
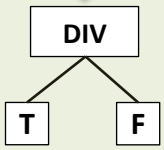
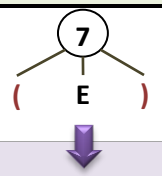
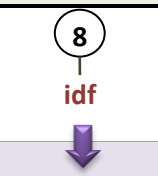
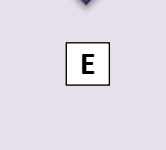
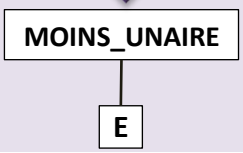
(2+3)×5$



L'arbre de dérivation ci-contre (correspondant au ***texte source*** ci-dessus) contient de nombreux nœuds inutiles pour l'analyse de l'expression arithmétique. On va simplifier au maximum cet arbre, afin d'obtenir un **arbre abstrait** tel que ci-dessous :



Pour réaliser cette simplification, on va associer à toute règle de la grammaire une action sémantique, c'est-à-dire un petit « bout de code » qu'on va exécuter au moment de la réduction de la règle pendant l'analyse syntaxique.

Non-terminal	Action sémantique pour chaque production		
E → *(arbre syntaxique)*			
E *(arbre abstrait)* devient…			
T → *(arbre syntaxique)*			
T *(arbre abstrait)* devient…			
F → *(arbre syntaxique)*			
F *(arbre abstrait)* devient…			

Chaque flèche correspond à une action sémantique. Les arbres sont par la même occasion renommés.

Chaque production de la grammaire d'arbre définit un opérateur à la racine et chaque opération a un profil.
Par exemple, les opérations **PLUS**, **MOINS**, **MULT** et **DIV** sont des opérations ***binaires***.
MOINS_UNAIRE et une opération ***unaire***. Enfin, **idf** est une opération ***zéro-aire***.

3) Utilisation du logiciel ANTLR

ANTLR (*ANother Tool for Language Recognition*) est un programme libre permettant de construire des compilateurs. **ANTLR** prend en entrée une grammaire définissant un langage de programmation et produit le code reconnaissant ce langage. Le logiciel permet de générer du code pour les langages suivants : *Ada95*, *ActionScript*, *C*, *C++*, *Java*, *JavaScript*, *Perl*, *Python*, *C#*, *Objective C*.
Il est possible de générer des analyseurs lexicaux, syntaxiques ou des analyseurs lexicaux et syntaxiques combinés. L'analyseur syntaxique généré est alors lui-même capable de générer automatiquement un **AST** après avoir lu le texte source, cet **AST** pouvant à son tour être traité par un analyseur d'arbre (un analyseur sémantique, *a priori*).

Le projet **OGAZ**, réalisé en cours de SD de première année, implémentait une petite partie de ces fonctionnalités.

Nous allons voir ici comment ajouter des actions sémantiques aux différentes productions d'une grammaire à l'aide du logiciel **ANTLR**.

Supposons qu'on dispose de la grammaire suivante, correspondant à une partie d'un langage de programmation (grammaire non complète). On autorise les expressions régulières dans la définition de la grammaire.

Grammaire:

program → declaration+
declaration → variable | fonction
variable → type ID ';'
type → 'int' | 'char'
fonction → type ID '(' (formalParam (',' formalParam)*) ? ')' block
formalParam →…
block →…

Rappels sur les expressions régulières :

- **a+** signifie "1 ou plusieurs **a**"
- **a*** signifie "0, 1 ou plusieurs **a**"
- **a?** signifie "0 ou 1 **a**"

Les **terminaux** sont ici placés entre guillemets simples.

Voici la notation (*la grammaire de grammaire*) utilisée par **ANTLR** pour définir la grammaire ci-dessus :

```
program :     declaration +
              ;
declaration : variable
              | fonction
              ;
variable :    type ID ';'
              ;
type :        'int'
              | 'char'
              ;
fonction :    type ID '(' (formalParam (',' formalParam)*) ? ')' block
              ;
etc.
```

On va passer en entrée d'**ANTLR** cette grammaire, et celui-ci va nous générer un programme capable d'analyser un texte source, et construire l'arbre abstrait si ce texte source respecte la grammaire. Pour l'instant, aucune action sémantique n'est associée aux règles de la grammaire. Par défaut, **ANTLR** va donc construire l'arbre de dérivation (l'arbre syntaxique « pur »).
Derrière chaque production, on ajoute les actions sémantiques sous la forme d'instructions compréhensibles par **ANTLR,** de ce genre : `→ ^(NOEUD feuille1 feuille2 …)`. Cela remplace la branche courante par une branche ayant pour nœud **NOEUD** et les feuilles `feuille1`, `feuille2`, …

Voici quelques exemples d'actions sémantiques pouvant être ajoutées dans la grammaire fournie à ANTLR :

- On peut renommer des éléments (nœuds ou feuilles) :

```
program :     declaration +
              → ^(ROOT declaration+)
              ;
```

Ici, on renomme le nœud "**program**" en "**ROOT**". Les déclarations restent les sous-nœuds de cette branche, elles sont ici conservées sans changement.

- On peut supprimer des éléments (nœuds ou feuilles) :

```
variable :    type ID ';'
              → ^(DECL type ID)
              ;
```

Ici, on supprime le non-terminal **';'**. De plus, on renomme le nœud "**variable**" en "**DECL**".

- On peut simplifier et réarranger des éléments (nœuds ou feuilles) :

```
fonction :    type ID '(' (formalParam (',' formalParam)*) ? ')' block
              → ^(FONC type ID formalParam* block)
              ;
```

Ici, la règle de grammaire est assez complexe, car il faut gérer correctement la possibilité d'avoir zéro, un ou plusieurs paramètres. Quand il n'y a qu'un seul paramètre, il ne faut pas qu'il y ait de virgule derrière celui-ci ; par contre, il en faut une entre chaque paramètre dès qu'il y en a au moins deux. Ainsi, cela donne une répétition du non-terminal `formalParam` dans la règle. L'action sémantique permet ici de supprimer les parenthèses et les virgules, ainsi que de regrouper les paramètres pour obtenir un arbre simplifié.

- Il est également possible de déclarer des variables dans les règles de grammaire d'**ANTLR**, pour s'en resservir dans les actions sémantiques (on fait précéder la variable du signe **$**). Pour connaître toutes les possibilités qu'offre ANTLR, il est nécessaire de consulter sa documentation, sur le site officiel : http://www.antlr.org/wiki/display/ANTLR3/Tree+construction

Contrôles sémantiques

Pour réaliser tout ou une partie des contrôles sémantiques, diverses manières de procéder sont possibles. Ces manières peuvent être utilisées conjointement, ou ne pas être utilisées.

Par exemple, certains langages autorisent la déclaration de variables non typées. C'est le cas du **C#** :

```
var i = 2 ;
```

Ici, malgré le fait qu'on ne déclare pas explicitement le fait que **i** est un entier (**int**), le compilateur **C#** est capable de le détecter « tout seul ». C'est ce qu'on appelle **l'inférence de type** : pour déduire le type d'une variable, le compilateur se sert de **règles d'inférence** ; cet ensemble de règles est donc une manière de réaliser une partie des contrôles sémantiques. Remarque : le langage **PHP** gère également l'inférence de type, bien qu'il ne s'agisse pas d'un langage compilé, mais interprété.

Il est également possible de se servir de **grammaires attribuées** (on associe aux terminaux, non-terminaux et/ou règles des informations permettant de réaliser des contrôles sémantiques).

Ces deux aspects des contrôles sémantiques (grammaires attribuées et règles d'inférence) **ne seront pas abordés dans ce cours**. La seule méthode que nous appliquerons pour réaliser l'ensemble des contrôles sémantiques est l'utilisation de la **représentation intermédiaire** disponible à la fin de la première passe : **l'arbre abstrait**.

1) Contrôles sémantiques *statiques* et contrôles sémantiques *dynamiques*

Deux types de contrôles sémantiques sont effectués :

- Les contrôles sémantiques statiques : ils constituent l'essentiel des contrôles effectués. Ils se déroulent à la **compilation**. C'est principalement ces contrôles dont nous allons traiter dans la suite de ce cours.
- Les contrôles sémantiques dynamiques : ce sont les contrôles qui ne peuvent être effectués à la compilation, et qui le sont par conséquent à **l'exécution**. Pour pouvoir être opérationnels, ces contrôles nécessitent donc l'ajout de code assembleur supplémentaire dans le programme cible. Par exemple, du code assembleur va être inséré avant une division, afin de vérifier s'il ne va pas y avoir une division par zéro (auquel cas le programme quitte « proprement » après avoir affiché un message d'erreur). Du code assembleur peut également être inséré dans le programme dans le but de vérifier si l'utilisateur accède à un élément d'un tableau dont l'indice est bien compris entre les bornes de ce tableau.

2) Représenter la correspondance utilisation/définition d'un identificateur

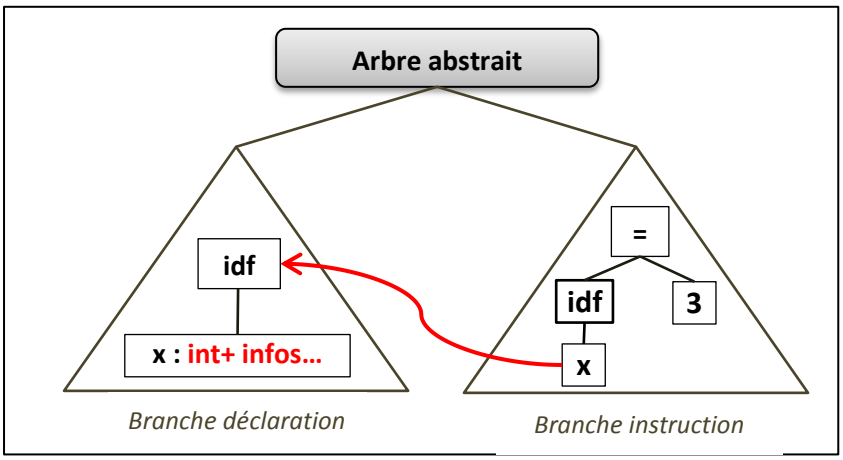
L'arbre abstrait obtenu à l'issue de l'analyse syntaxique se divise en deux branches principales :

- **une branche déclaration correspondant aux déclarations** de variables, fonctions, etc. Chacune des variables déclarées dans le programme est donc présente dans ce sous-arbre.
- **une branche instruction correspondant aux affectations** de variables, utilisation de fonctions, etc. Chaque variable citée dans cette branche doit bien sûr être présente dans la branche déclaration.

Une des fonctions principales de l'analyseur sémantique va être de vérifier que chaque variable utilisée est correctement déclarée. Pour cela, il faut analyser l'arbre abstrait afin de trouver et d'établir une correspondance entre une utilisation d'identificateur, et sa déclaration. Nous allons étudier **trois méthodes** pour ce faire.

Méthode n°1

Cette première méthode va consister à décorer la branche déclaration de l'arbre : à chaque variable on associe des informations : type d'identificateur (variable, fonction), type de variable, bloc de déclaration, portée dans le programme, etc. Ainsi, les « décorations » correspondent à ces informations supplémentaires ajoutées dans l'arbre directement.



Exemple pour le code suivant :

```
int x ;
x = 3 ;
```

Chaque variable utilisée dans la branche instruction est liée à sa déclaration dans la branche déclaration. Les informations ajoutées par l'analyseur sémantique sont **en rouge**.

Cette méthode a l'inconvénient de produire un arbre assez lourd, et plus difficilement lisible par le générateur de code lors de la troisième passe.

Méthode n°2

La deuxième méthode consiste à faire en sorte que le générateur de code ne se serve plus de la branche déclaration pour produire le code cible, mais qu'il utilise à la place une **table des symboles** (**TDS**). Cette table des symboles sera créée au début de l'analyse sémantique, et remplie au fur et à mesure de cette analyse. On crée généralement une table des symboles par région ; à la fin de l'analyse, on rassemble le tout dans une table des symboles globale.

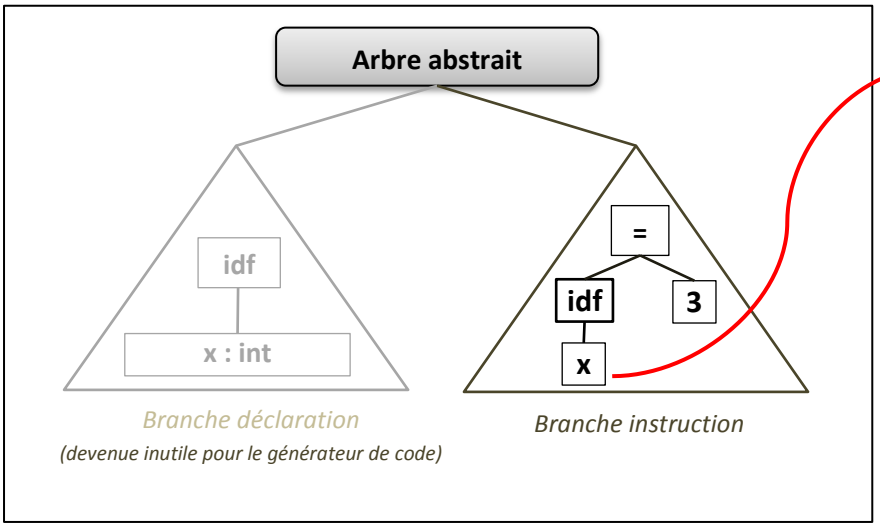


Table des symboles (TDS)	
x	de type **int** + **infos**…
…	…

L'exemple précédent est repris. Ce qui est en **noir** (et grisé) correspond à ce qu'on obtient à la fin de la première passe (l'arbre abstrait, qui va être parcouru lors de la seconde passe). Ce qui est en **rouge** correspond à ce qu'on a ajouté lors de la deuxième passe. Enfin, ce qui est grisé correspond à ce dont on a plus besoin pour la troisième passe.

Méthode n°3

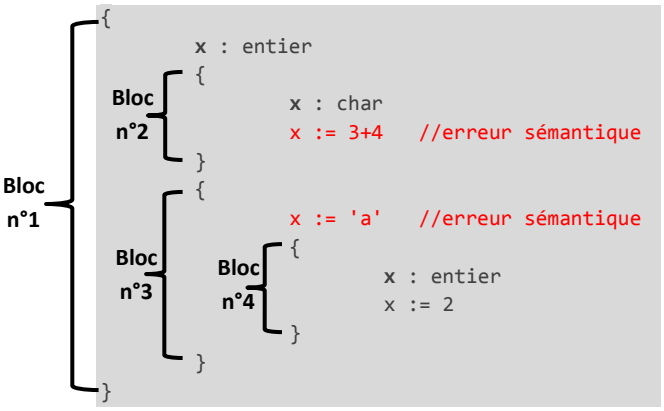
La troisième méthode est très proche de la seconde ; la principale différence réside dans le fait que la table des symboles est créée **dès l'analyse syntaxique** (et pas après). Ainsi, l'analyse syntaxique s'occupe toujours de générer l'arbre abstrait, mais s'occupe également de pré-remplir la table des symboles. A l'issue de l'analyse syntaxique, on a donc une table des symboles en partie remplie. L'analyse sémantique s'occupera alors de la compléter.

3) Précisions sur la table des symboles (TDS)

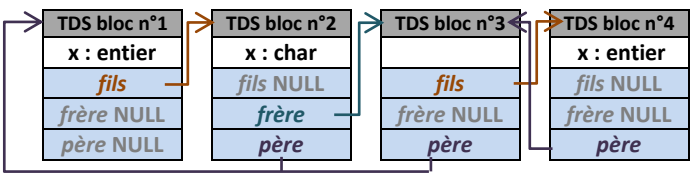
Généralement, on crée une **table des symboles par région** (ou bloc) : par fonction, procédure, bloc sans nom, …
Une table des symboles contient une entrée par identificateur. Cette entrée contient les informations suivantes :

- le type, la place mémoire occupée, et les bornes s'il s'agit d'un tableau
- le déplacement (différence d'adresses mémoires) nécessaire pour accéder à la variable depuis la base de la pile ou de la fonction (*voir chapitres 5 et 6*)
- la visibilité et la portée de cet identifiant (numéro de bloc, nombre d'imbrications)
- s'il s'agit d'une fonction, les arguments et leur type, ainsi que le type de retour, sont également précisés.

Exemple :



La numérotation des blocs ainsi que la connaissance des imbrications sont réalisées grâce à la pile des blocs ouverts. A l'issue de l'analyse syntaxique (méthode n°3) ou sémantique (méthode n°2), on a une TDS par bloc :



De plus on regroupe les différentes TDS en une **TDS globale**. Il y a autant d'entrées que de variables locales (ici), et à chaque variable on associe le numéro de bloc.

Chapitre n°5 : représentation des objets en mémoire à l'exécution

Introduction

Un processus est un programme en cours d'exécution par un ordinateur. On peut le définir plus précisément par :

- **un ensemble d'instructions à exécuter** (un programme, en langage machine), parfois situé dans une mémoire morte (fréquent dans les systèmes embarqués notamment), mais le plus souvent chargé depuis une mémoire de masse (disque dur) vers la mémoire vive (RAM);
- un **espace d'adressage en mémoire vive** pour stocker toutes **les données de travail** (variables, adresses mémoire, objets, …) nécessaires au bon déroulement de l'exécution du programme. Cet espace se divise en deux parties principales : la **pile d'exécution** (*call stack*) et le **tas** (*heap*).

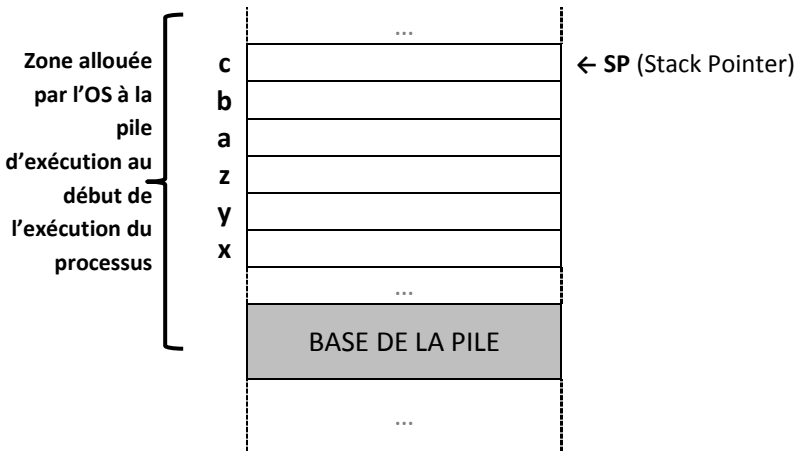
La **pile d'exécution** (ou simplement **pile**) sert à enregistrer des informations relatives à une fonction en train d'être exécutée dans le programme. Elle est utilisée pour emmagasiner plusieurs valeurs : les variables locales de la fonction, les paramètres de la fonction, mais aussi et surtout la trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution (l'adresse de retour). Dans un registre du processeur (par exemple, **R31**) est enregistré **le pointeur du sommet de pile** (*Stack Pointer*, ***SP***), qui correspond à l'adresse mémoire où se situe le sommet de la pile. Ainsi, pendant l'exécution, quand on empile des données dans la pile ou qu'on dépile, il faut mettre à jour la valeur de ce pointeur. Si jamais l'accumulation de données consomme tout l'espace alloué à la pile d'exécution (par exemple, avec une fonction récursive qui se rappelle à l'infini), un message d'erreur appelé un **dépassement de pile** se produit (en **Java**, cela correspond au *StackOverflow Error*).

Le **tas** est utilisé pour allouer dynamiquement de l'espace mémoire à la demande du programme, par opposition à la pile qui est gérée automatiquement lors d'un appel de sous-routine ou de fonction. Il sert souvent à stocker des données de plus grande taille (tableaux, structures, et objets). L'allocation et la libération de la mémoire dans le tas est généralement sous la responsabilité du programmeur. En **C**, ces deux actions peuvent être réalisées grâce aux instructions ***malloc*** et ***free***, respectivement. En **Java**, l'allocation est réalisée grâce au mot-clé ***new*** ; en revanche, la libération de la mémoire est automatisée grâce au ***ramasse-miette*** (*garbage collector*).

Exemple avec schéma :

```
int main(){
      int x,y,z ;
      char a,b,c ;
}
```

Voici à l'exécution **l'état de la pile** juste après la dernière instruction :



Principaux types de variables en mémoire

1) Types simples

On y trouve les variables les plus communes (***boolean***, ***byte***, ***char***, ***double***, ***float***, ***int***, ***long***, ***short***). On y trouve également le type **pointeur** (c'est-à-dire une adresse mémoire).
Elles sont stockées dans la pile où elles y prennent une place donnée. Généralement, leur taille n'excède pas 16 octets. Cela dépend néanmoins du langage et de la machine utilisée (un entier de type **int** prend 4 octets sur une machine 32 bits, et 8 octets sur une machine 64 bits).

2) Types énumérés

On définit une liste de constantes. Par exemple en **C** : `enum Jour {DIMANCHE, LUNDI, …};`
En réalité, il s'agit juste de « simplifier la vie » du programmeur : à chaque constante est associé un entier (donc un type simple) : **DIMANCHE** est équivalent à **0**, **LUNDI** à **1**, etc. Toute variable déclarée comme étant de type **Jour** n'est donc rien d'autre qu'un **int**.

3) Enregistrements

Un enregistrement permet de créer ses propres types de variables. Il s'agit la plupart du temps d'un type complexe (plusieurs types simples mis bout-à-bout). Par exemple en **C :**

```
struct Couleur {
      int r;
      int g;
      int b;
};
struct Couleur p = {64, 255, 128};
```

"struct Couleur" est un nouveau type pour le langage ; ici la variable **p** est de ce type. Cette manière de créer la variable **p** (en utilisant les **accolades** et en mettant les valeurs des différents champs les uns après les autres) fait qu'elle se situera entièrement dans la pile d'exécution. Le tas ne sera pas utilisé, à moins de remplacer la notation en accolades par un ***malloc***. Dans ce second cas, les valeurs des champs seront dans le tas mais la pile contiendra néanmoins un pointeur (une adresse mémoire, donc un type simple) vers le premier de ces champs, ainsi qu'une valeur correspondant à la taille utilisée par la structure dans le tas.

Il peut être préférable de créer la structure dans le tas (en utilisant donc ***malloc***), car sinon la structure restera locale à la fonction en cours d'exécution : une fois cette fonction terminée, la pile d'exécution sera dépilée et les valeurs des champs de la structure risqueront d'être écrasés si une autre fonction est appelée.

4) Unions

Les unions permettent de dire au compilateur qu'une variable peut être vue comme étant d'un type particulier, *ou bien* d'un autre type. Par exemple en **C** : `union intOuFloat {int i; float x;} var;`
Cette instruction permet d'avoir une variable (ici, nommée **var,** de type **intOuFloat**) pouvant être vue comme un entier (dans ce cas, on l'appelle en faisant **var.i**) ou bien comme un flottant (en l'appelant **var.x**). Du point de vue de la mémoire, la taille occupée par cette variable sera la taille du plus grand champ (ici *float*).
Les variables de ce type sont également stockées dans la pile.

Type "complexe" : le cas des tableaux

Les tableaux sont un cas particulier des types complexes. On entend par tableau la mise bout-à-bout d'un certain nombre d'éléments dans la mémoire, tous du même type simple. Les tableaux peuvent être de plusieurs dimensions. Quand un tableau possède plusieurs dimensions (au moins 2), il existe deux moyens de ranger les éléments : de manière contigüe, ou bien par indexations successives.

1) Principes généraux

On peut différencier trois principaux types de tableaux :

- **les tableaux statiques :** la taille est fixe, et est connue dès la compilation. Cette taille est donc codée « en dur » (sous la forme d'une constante) au moment de la déclaration du tableau dans le code source.

  Exemple en **C** : `int tab[5];`

- **les tableaux dynamiques :** la taille est fixe, mais connue qu'à l'exécution. Ainsi lors de la déclaration du tableau, la taille est précisée depuis une variable.

  Exemple en **C** : `int n = 2 + a;  int tab[n];` *où **a** est une autre variable de type **int***

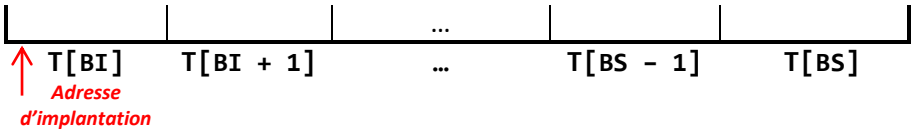
- **les tableaux flexibles :** la taille est variable à l'exécution. En général, dans un langage comme le **C**, il est nécessaire de procéder à une réallocation mémoire quand le tableau change de taille.

Dans tous les cas, le nombre de dimensions du tableau est connu à la compilation.

2) Implantation en mémoire des tableaux à une dimension

On dispose d'un tableau T à une dimension dont la borne inférieure (indice du premier élément) est appelée **BI** et la borne supérieure (indice du dernier élément) **BS**. On notera : **T[BI..BS]** (**BS ≥ BI**)

Remarque : En **C**, ainsi que dans beaucoup d'autres langages de programmation, la borne inférieure est **0** (premier élément du tableau). La borne supérieure correspondra donc à la dimension du tableau moins un.



Adresse d'implantation

L'adresse d'implantation (**@impl**) est l'adresse du premier élément du tableau. Elle n'est connue qu'à l'exécution.

La **place mémoire** nécessaire pour un tableau à une dimension vaut : **(BS-BI+1)×t**.

Etant donné un indice **i** compris entre **BI** et **BS**, on souhaite connaître l'adresse de l'élément **T[i]**, en fonction de **@impl**. On connaît également la taille **t** prise par chaque élément du tableau.

$$@T[i] = @impl + (i - BI)\times t = @impl - (BI\times t) + i\times t$$

BI et **t** sont deux valeurs connues dès la compilation, on peut donc calculer **BI×t** dès la compilation et ranger cette valeur dans la table des symboles.

On définit **l'origine virtuelle** par **@OV = @impl – (BI×t)**

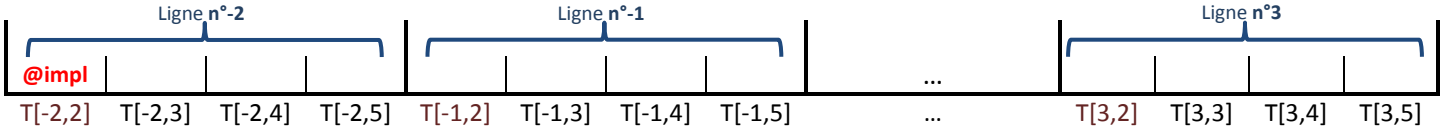
Cette valeur peut être calculée une fois pour toutes lors de la création du tableau, à l'exécution du programme. Elle est indépendante de l'indice de l'élément recherché dans le tableau. Ainsi, à un moment donné de l'exécution du programme, on accèdera au **i**ème élément du tableau en calculant la somme **@OV + i×t.**

3) Tableaux à plusieurs dimensions : implantation contigüe

Il faut préciser les bornes pour chaque dimension du tableau. Par exemple pour un tableau à deux dimensions, on notera : **T[BI_1..BS_1 , BI_2..BS_2]**, où **BI_1**, **BS_1** sont les bornes de la première dimension, **BI_2**, **BS_2** celles de la deuxième.

Exemple :

Soit **T[-2..3 , 2..5]** un tableau d'entiers. L'implantation contigüe nous donne une représentation mémoire de ce type :



Cas d'un tableau à deux dimensions

On suppose que le rangement est fait par ligne (les sous-tableaux sont déterminés d'après la première dimension). La taille **t** d'un élément est connue. Calculons dans ce cas l'adresse mémoire de l'élément à l'emplacement **(i, j)** :

$$@T[i,j] = @impl + (i-BI_1)\times(BS_2-BI_2+1)\times t + (j-BI_2)\times t$$

$$= \underbrace{(@impl - BI_1\times(BS_2-BI_2+1)\times t - BI2\times t)}_{\text{Origine virtuelle (@OV)}} + i\times\underbrace{(BS_2-BI_2+1)\times t}_{\text{enjambée n°1}} + \underbrace{j*t}_{\text{enjambée n°2}}$$

L'origine virtuelle, ainsi que les deux enjambées (notées e_1 et e_2) sont indépendantes des indices **i** et **j**.

Calcul des différentes valeurs de l'expression : à la compilation ou à l'exécution ?

On sait que :

- **@impl** n'est connue qu'à l'exécution, lors de la création du tableau
- les bornes BI_i, BS_i, ainsi que la taille **t** des éléments sont connues dès la compilation pour les **tableaux statiques**
- les bornes BI_i, BS_i, ne sont connues qu'à l'exécution, pour les **tableaux dynamiques et flexibles**
- les indices **i** et **j** ne sont connus qu'à l'exécution, au moment où on souhaite accéder à l'élément **(i, j)** du tableau.

Par conséquent :

- pour les **tableaux statiques** la valeur $v = BI_1\times(BS_2-BI_2+1)\times t + BI2\times t$, ainsi que les enjambées e_1 et e_2, peuvent être calculées à la compilation et rangées dans la TDS.
- pour les **tableaux dynamiques et flexibles**, ces valeurs sont calculées à l'exécution et rangées dans la pile.
- pour les **tableaux flexibles**, ces valeurs sont recalculées dès qu'une borne change.
- **l'origine virtuelle** dépend de **@impl**, elle n'est donc calculée qu'à l'exécution. Son calcul (**@impl – v**) nécessite donc l'ajout de code assembleur supplémentaire dans le code à générer.

Du fait de sa taille variable, un **tableau flexible** est obligatoirement rangé dans le tas (et non dans la pile d'exécution).