كلـيـــــــــــة العلوم
السملالية - مراكش
FACULTE DES SCIENCES
SEMLALIA-MARRAKECH

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD

Département Informatique

Licence SMI – S6

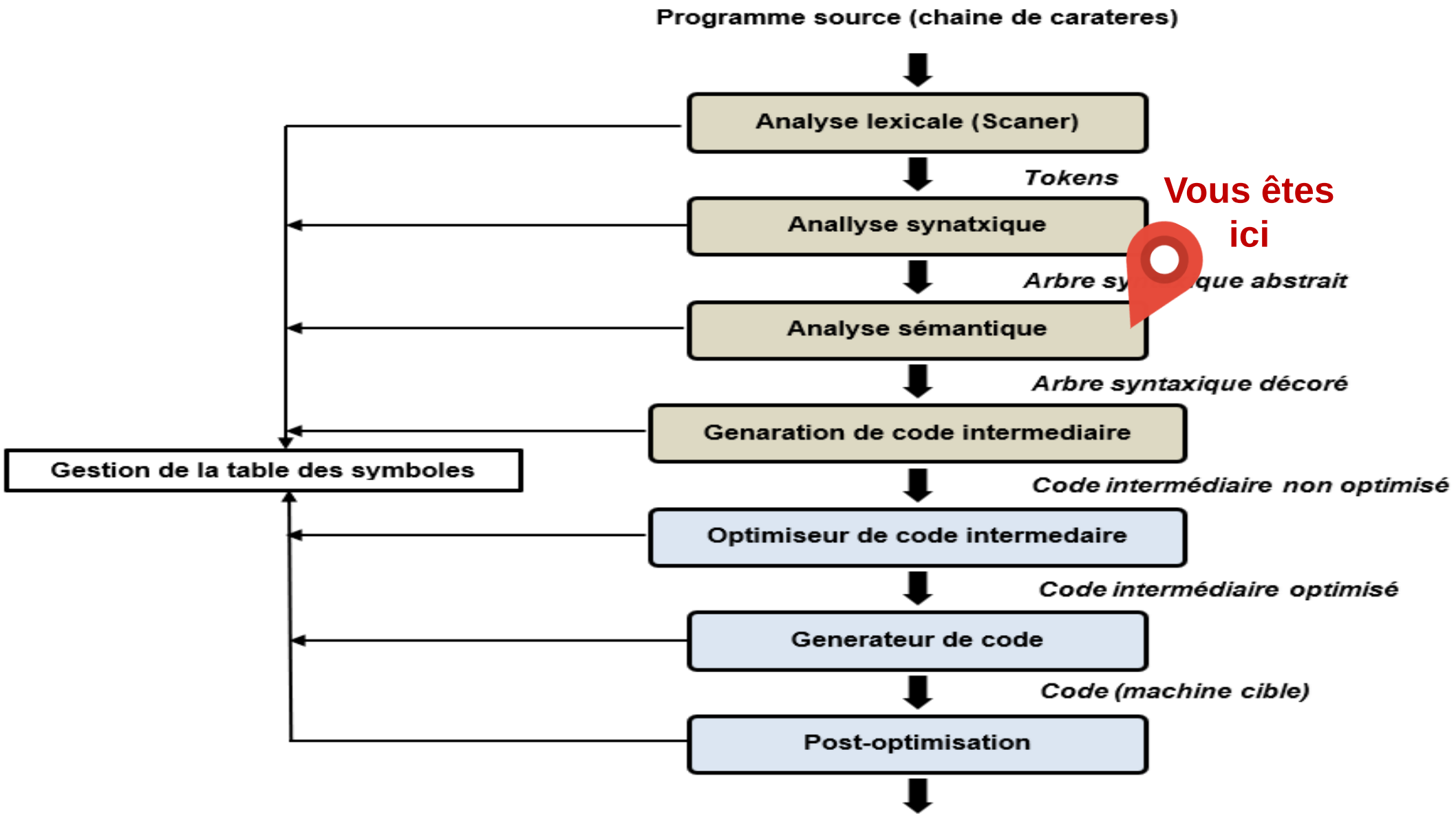
Compilation

Pr. Aimad QAZDAR

aimad.qazdar@uca.ac.ma

30 avril 2021

Rappel : Phases d'un compilateur



Chapitre 4:
Analyse sémantique

Exemple

```
int main (){
   int a ;
   a=0;
   a =a+b;
}
```

```
int main (){
   int a , b=0;
   int a=0;
   a =a+b;
}
```

Plan

Introduction

- Après l'analyse lexicale et l'analyse syntaxique, l'étape suivante dans la conception d'un compilateur est l'analyse sémantique dont la partie la plus visible est le **contrôle de type.**
- Quelques tâches liées au contrôle de type sont :
  - Construire et mémoriser des représentations des types définis par l'utilisateur, lorsque le langage le permet ;
  - Traiter les déclarations de variables et fonctions et mémoriser les types qui leur sont appliqués ;
  - Vérifier que toute variable référencée et toute fonction appelée ont bien été préalablement déclarées ;
  - Vérifier que les paramètres des fonctions ont les types requis ;
  - Contrôler les types des opérandes des opérations arithmétiques et en déduire le type du résultat ;
  - etc.

Introduction

- Imaginons qu'un programme, écrit en C, contient l'instruction :

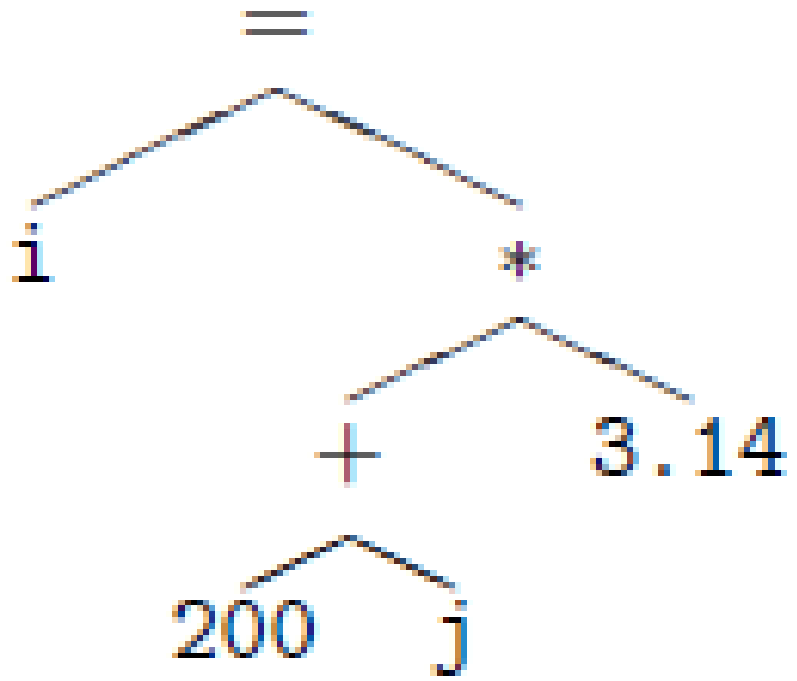
  i = (200 + j) * 3.14 .

- L'analyseur syntaxique construit un *arbre abstrait* représentant cette expression, comme ceci :

  – Pour être tout à fait corrects, à la place de i et j nous aurions dû représenter des renvois à la table des symboles

  – Seules les feuilles ont des **types précis**

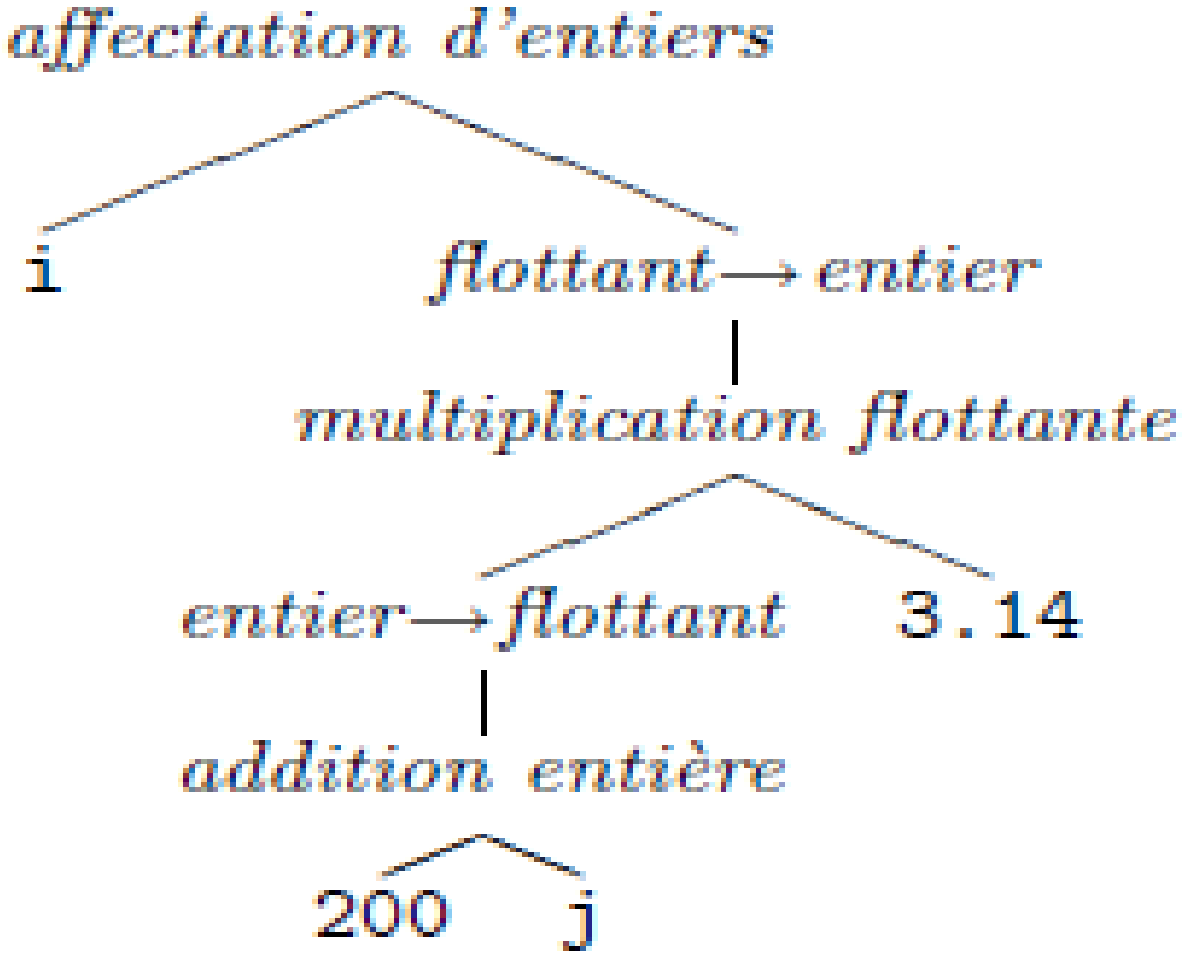
  – Les nœuds internes représentent des opérations abstraites, dont le **type exact reste à préciser**



Le travail sémantique à faire consiste à « remonter les types », depuis les feuilles vers la racine, rendant concrets les opérateurs et donnant un type précis aux sous-arbres.

- Supposons par exemple que i et j aient été déclarées de type entier.
- L'analyse sémantique de l'arbre précédent permet d'en déduire, successivement :
  - que le + est l'addition des entiers, puisque les deux opérandes sont entiers, et donc que le sous-arbre chapeauté par le + représente une valeur de type entier ;
  - que le * est la multiplication des flottants, puisqu'un opérande est flottant, qu'il y a lieu de convertir l'autre opérande vers le type flottant « **règle du plus fort** » , et que le sous-arbre chapeauté par * représente un objet de type flottant ;
  - que l'affectation au sommet de l'arbre consiste donc en l'affectation d'un flottant à un entier, et qu'il faut donc insérer une opération de **troncation** flottant → entier *;* en C, il en découle que l'arbre tout entier représente une valeur du type entier.

- Le contrôle de type transforme l'arbre précédent en quelque chose qui ressemble à ceci :



Représentation et reconnaissance des types

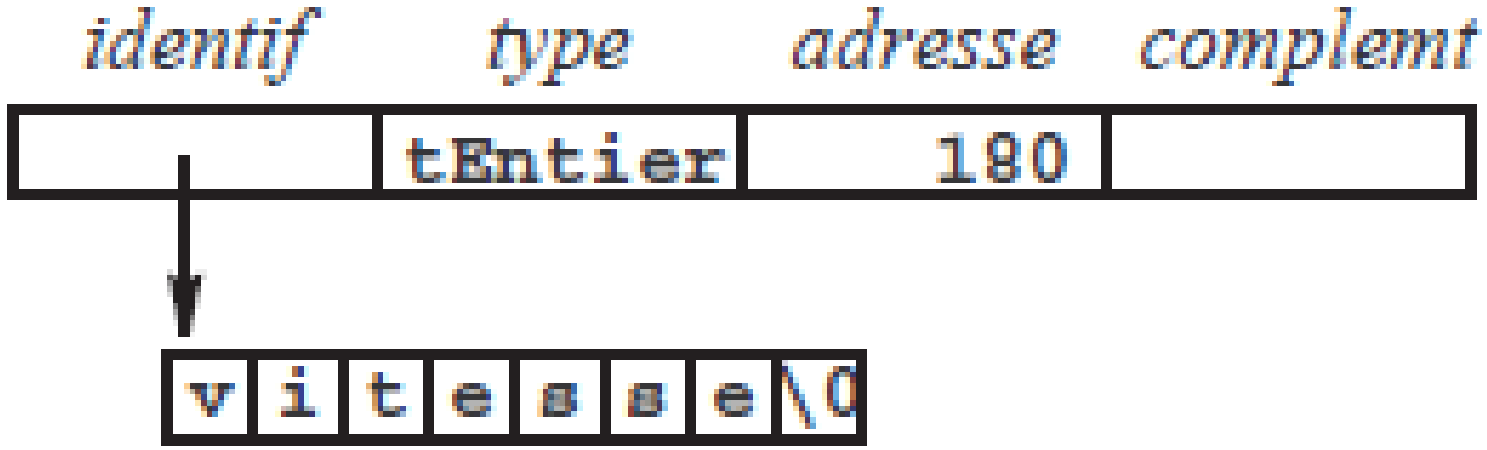
- Une partie importante du travail sémantique qu'un compilateur fait sur un programme est :
  - Pendant la compilation des déclarations, construire des représentations des types déclarés dans le programme ;
  - Pendant la compilation des instructions, reconnaître les types des objets intervenant dans les expressions.
- La principale difficulté de ce travail est la complexité des structures à construire et à manipuler.
- En effet, dans les langages modernes, les types sont définis par des procédés récursifs qu'on peut composer à volonté.

Représentation et reconnaissance des types

- Par exemple, en C on peut avoir des entiers, des adresses (ou pointeurs) d'entiers, des fonctions rendant des adresses d'entiers, des adresses de fonctions rendant des adresses d'entiers, des tableaux d'adresses de fonctions rendant des adresses d'entiers, etc.
- Cela peut aller aussi loin qu'on veut, l'auteur d'un compilateur doit se donner le moyen de représenter ces structures de complexité quelconque.

Dictionnaire

- Dans les langages de programmation les variables et les fonctions doivent être déclarées avant d'être utilisées dans les instructions.
- Quel que soit le degré de complexité des types supportés par notre compilateur, celui-ci devra gérer une ***table de symboles***, appelée aussi ***dictionnaire***, dans laquelle se trouveront les identificateurs couramment déclarés,
- Chacun identificateur déclaré est associé à certains attributs, comme son *type*, son *adresse* et d'autres informations,



Fonctionnement du dictionnaire

- Quand le compilateur trouve un **identificateur dans une déclaration**, il le **cherche dans le dictionnaire en espérant ne pas le trouver** (sinon c'est l'erreur « identificateur déjà déclaré »), puis il l'ajoute au dictionnaire avec le type que la déclaration spécifie ;
- Quand le compilateur trouve un **identificateur dans la partie exécutable** d'un programme, il le **cherche dans le dictionnaire avec l'espoir de le trouver** (sinon c'est l'erreur « identificateur non déclaré »), ensuite il utilise les informations que le dictionnaire associe à l'identificateur.

Type du dictionnaire

- Un programme est essentiellement une collection de fonctions, entre lesquelles se trouvent des déclarations de variables.
- À l'intérieur des fonctions se trouvent également des déclarations de variables.
- Les variables déclarées entre les fonctions et les fonctions elles-mêmes sont des *objets globaux*.
- Un **objet global** est visible depuis sa déclaration jusqu'à la fin du texte source, sauf aux endroits où un objet local de même nom le masque.
- Les variables déclarées à l'intérieur des fonctions sont des ***objets locaux***.
- Un objet local est visible dans la fonction où il est déclaré, depuis sa déclaration jusqu'à la fin de cette fonction; il n'est pas visible depuis les autres fonctions
- En tout point où il est visible, un objet local *masque* tout éventuel objet global qui aurait le même nom.

Type du dictionnaire

- Quand le compilateur se trouve dans une fonction, il faut posséder deux dictionnaires :
  - un ***dictionnaire global***, contenant les noms des objets globaux couramment déclarés,
  - un ***dictionnaire local*** dans lequel se trouvent les noms des objets locaux couramment déclarés (qui, parfois, masquent des objets dont les noms se trouvent dans le dictionnaire global).

Type du dictionnaire

- **Cas 1 :** Lorsque le compilateur traite la déclaration d'un identificateur *i* qui se trouve à l'intérieur d'une fonction,
  - *i* est recherché dans le dictionnaire local *exclusivement ;*
  - Normalement, il ne s'y trouve pas (sinon, « erreur : identificateur déjà déclaré »).
  - Suite à cette déclaration, *i* est ajouté au dictionnaire local.
  - Il n'y a strictement aucun intérêt à savoir si i *fi*gure à ce moment-là dans le dictionnaire global.
- **Cas 2:** Lorsque le compilateur traite la déclaration d'un identificateur i en dehors de toute fonction,
  - i est recherché dans le dictionnaire global, qui est le seul dictionnaire existant en ce point ;
  - Normalement, il ne s'y trouve pas (sinon, « erreur : identificateur déjà déclaré »).
  - Suite à cette déclaration, i est ajouté au dictionnaire global.

Type du dictionnaire

- **Cas 3** : Lorsque le compilateur compile une instruction exécutable, forcément à l'intérieur d'une fonction,
  - Chaque identificateur *i* rencontré est recherché d'abord dans le dictionnaire local ;
  - S'il ne s'y trouve pas, il est recherché ensuite dans le dictionnaire global
  - Si les deux recherches échouent, « erreur : identificateur non déclaré ».
  - En procédant ainsi on assure le masquage des objets globaux par les objets locaux.
- **Cas 4** : Lorsque le compilateur quitte une fonction,
  - Le dictionnaire local en cours d'utilisation est détruit, puisque les objets locaux ne sont pas visibles à l'extérieur de la fonction.
  - Un dictionnaire local nouveau, vide, est créé lorsque le compilateur entre dans une fonction.

Type de dictionnaire

Tout au long d'une compilation le dictionnaire global ne diminue jamais

A l'intérieur d'une fonction le dictionnaire global n'augmente pas

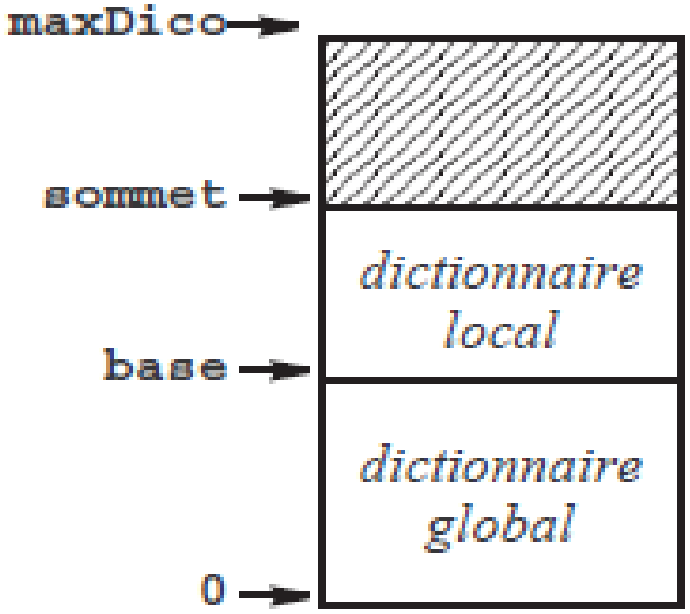
Le dictionnaire global n'augmente que lorsque le dictionnaire local n'existe pas

Implémentation de dictionnaire

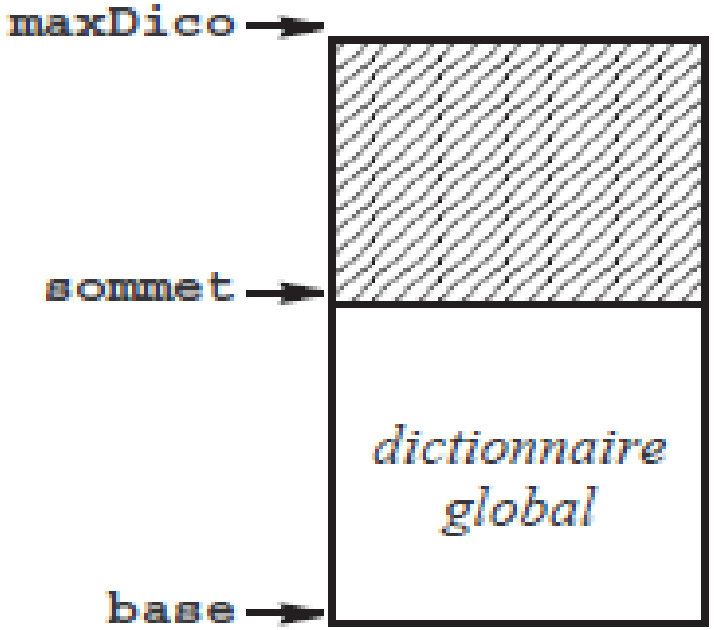
- **Tableau à accès séquentiel**
- **Tableau trié er recherche dichotomie**
- **Arbre binaire de recherche**
- **Adressage dispersé**

Tableau à accès séquentiel

- L'implémentation la plus simple des dictionnaires consiste en un tableau dans lequel les identificateurs sont placés dans l'ordre où leurs déclarations ont été trouvées dans le texte source.
- Dans ce tableau, les recherches sont séquentielles.
- Lorsqu'il existe, le dictionnaire local se trouve au-dessus du dictionnaire global (en supposant que le tableau grandit du bas vers le haut).



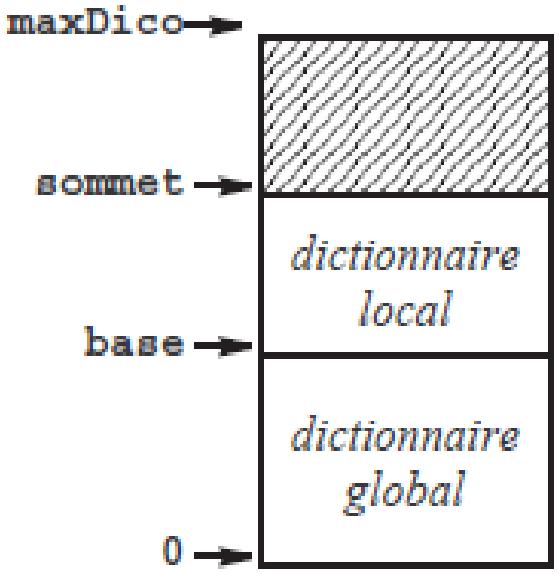
*Dictionnaire quand on est à **l'intérieur** des fonctions*



*Dictionnaire quand on est à **l'extérieur** des fonctions*

Tableau à accès séquentiel

- Trois variables sont essentielles dans la gestion du dictionnaire :
  - **maxDico** est le nombre maximum d'entrées possibles (à ce propos, voir « Augmentation de la taille du dictionnaire » un peu plus loin) ;
  - **sommet** est le nombre d'entrées valides dans le dictionnaire ; on doit avoir sommet ≤ maxDico ;
  - **base** est le premier élément du dictionnaire du dessus (c'est-à-dire le dictionnaire local quand il y en a deux, le dictionnaire global quand il n'y en a qu'un)



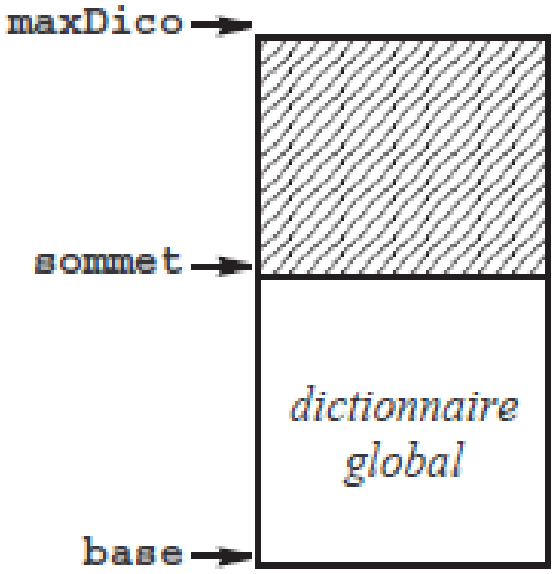


Tableau à accès séquentiel

- Les opérations nécessaires à la manipulation du dictionnaire sont :
  - **Recherche d'un identificateur pendant le traitement d'une déclaration** (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction) : rechercher l'identificateur dans la portion de tableau comprise entre les bornes **sommet - 1** et **base** ;
  - **Recherche d'un identificateur pendant le traitement d'une expression exécutable** : rechercher l'identificateur en parcourant *dans le sens des indices décroissants* la portion de tableau comprise entre les bornes **sommet - 1** et **0** ;
  - **Ajout d'une entrée dans le dictionnaire** (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction) : après avoir vérifié que **sommet < maxDico**, placer la nouvelle entrée à la position ***sommet***, puis faire **sommet** ← **sommet + 1** ;
  - **Création d'un dictionnaire local**, au moment de l'entrée dans une fonction :

    faire **base** ← **sommet**,
  - **Destruction du dictionnaire local**, à la sortie d'une fonction :

    faire **sommet** ← **base** puis **base** ← **0**.

Problème d’augmentation de la taille du dictionnaire

- Une question technique assez agaçante qu'il faut régler lors de l'implémentation d'un dictionnaire par un tableau est **le choix de la taille à donner à ce tableau.**
- Etant entendu qu'on ne connaît pas à l'avance la grosseur (en nombre de déclarations) des programmes que notre compilateur devra traiter.
- La bibliothèque C offre un moyen pratique pour résoudre ce problème, la fonction ***realloc*** qui permet d'augmenter la taille d'un espace alloué dynamiquement tout en préservant le contenu de cet espace.

Exemple

- La déclaration et les fonctions de gestion d'un dictionnaire réalisé dans un tableau ayant au départ la place pour 50 éléments ;
- Chaque fois que la place manque, le tableau est agrandi d'autant qu'il faut pour loger 25 nouveaux éléments :

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef
struct {
    char *identif;
    int type;
    int adresse;
    int complement;
} ENTREE_DICO;

#define TAILLE_INITIALE_DICO 50
#define INCREMENT_TAILLE_DICO 25

ENTREE_DICO *dico;
int maxDico, sommet, base;
```

```c
void creerDico(void) {
    maxDico = TAILLE_INITIALE_DICO;
    dico = malloc(maxDico * sizeof(ENTREE_DICO));
    if (dico == NULL)
        erreurFatale("Erreur interne (pas assez de mémoire)");
    sommet = base = 0;
}

void agrandirDico(void) {
    maxDico = maxDico + INCREMENT_TAILLE_DICO;
    dico = realloc(dico, maxDico);
    if (dico == NULL)
        erreurFatale("Erreur interne (pas assez de mémoire)");
}

void erreurFatale(char *message) {
    fprintf(stderr, "%s\n", message);
    exit(-1);
}
```

Exemple

- La fonction qui ajoute une entrée au dictionnaire :

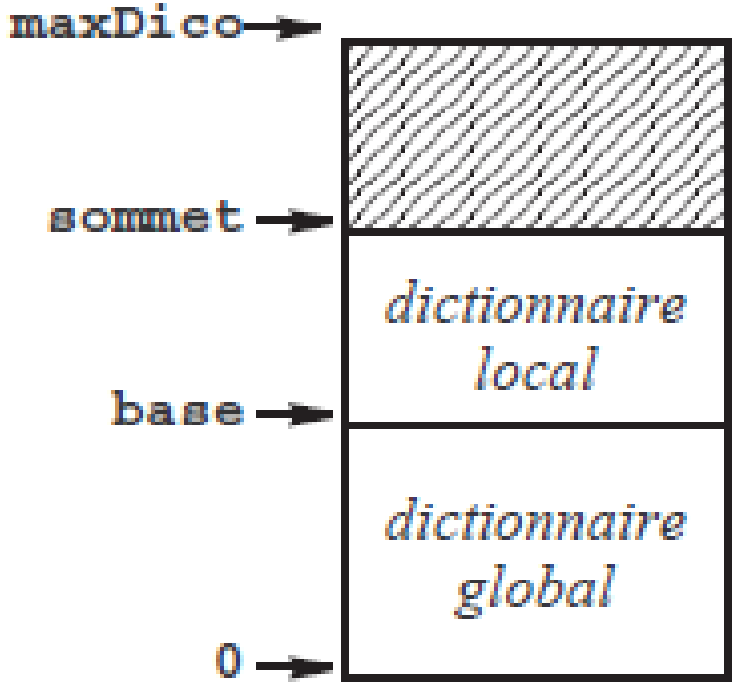
```
1  void ajouterEntree(char *identif, int type, int adresse, int complement) {
2      if (sommet >= maxDico)
3          agrandirDico();
4
5      dico[sommet].identif = malloc(strlen(identif) + 1);
6      if (dico[sommet].identif == NULL)
7          erreurFatale("Erreur interne (pas assez de mémoire)");
8      strcpy(dico[sommet].identif, identif);
9      dico[sommet].type = type;
10     dico[sommet].adresse = adresse;
11     dico[sommet].complement = complement;
12     sommet++;
13 }
```

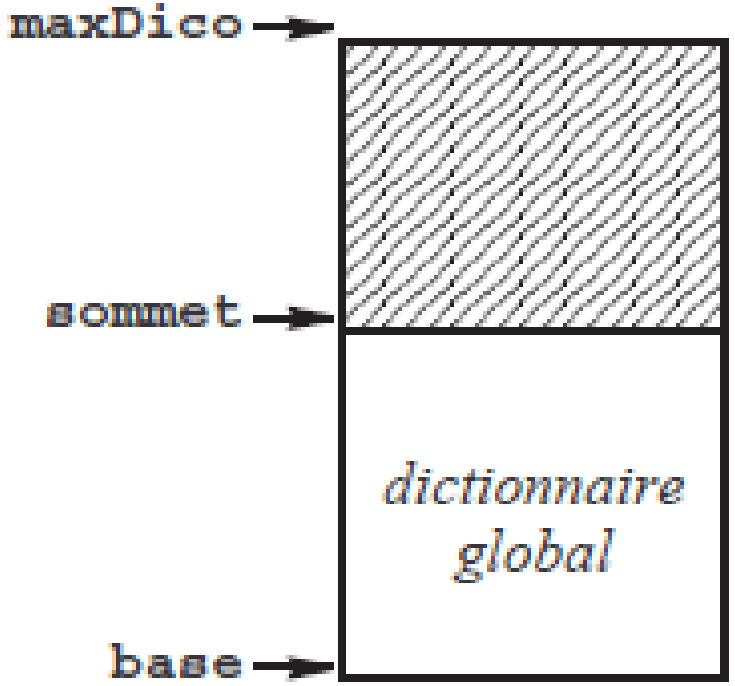
Tableau à accès séquentiel

- L'implémentation des dictionnaires par les tableaux à accès séquentiel est facile à mettre en œuvre et suffisante pour des applications simples, mais pas très efficace.
- Pratiquement on recherche des implémentations plus efficaces, car un compilateur passe beaucoup de temps à rechercher des identificateurs dans les dictionnaires.
- On estime que plus de 50 % du temps d'une compilation est dépensé en recherches dans les dictionnaires.
- Une première amélioration des dictionnaires consiste à maintenir des **tableaux triés**, permettant des **recherches par dichotomie**

Tableau trié et recherche dichotomique

- Les éléments d'indices allant de **base** à **sommet - 1**
- Lorsqu'il y a lieu, ceux d'indices allant de **0** à **base - 1**, sont placés en ordre croissant des identificateurs.





- La fonction **exist()** pour vérifier l'existence d'un identif dans la table
- La fonction **ajoutEntree()** pour ajouter un idenfi. dans la table

Tableau trié et recherche dichotomique

- La fonction **existe()**
  - Effectue la recherche de l'identificateur représenté par ident dans la partie du tableau trié, comprise entre les indices inf et sup, bornes incluses.
  - Le résultat de la fonction (1 ou 0, interprétés comme vrai ou faux) est la réponse à la question « l'élément cherché se trouve-t-il dans le tableau ? ».
- En outre, au retour de la fonction, la variable pointée par ptrPosition contient la position de l'élément recherché.
- C.-à-d.:
  - si l'identificateur est dans le tableau, l'indice de l'entrée correspondante ;
  - si l'identificateur ne se trouve pas dans le tableau, l'indice auquel il faudrait insérer, les cas échéant, une entrée concernant cet identificateur.

Tableau trié et recherche dichotomique

```
1  int existe(char *identif, int inf, int sup, int *ptrPosition) {
2      int i, j, k;
3
4      i = inf;
5      j = sup;
6      while (i <= j) {        /* invariant: i <= position <= j + 1 */
7          k = (i + j) / 2;
8          if (strcmp(dico[k].identif, identif) < 0)
9              i = k + 1;
10         else
11             j = k - 1;
12     }
13         /* ici, de plus, i > j, soit i = j + 1 */
14     *ptrPosition = i;
15     return i <= sup && strcmp(dico[i].identif, identif) == 0;
16 }
17
```

Tableau trié et recherche dichotomique

```
1  void ajouterEntree(int position, char *identif, int type, int adresse, int complt) {
2      int i;
3      if (sommet >= maxDico)
4          agrandirDico();
5      for (i = sommet - 1; i >= position; i--)
6          dico[i + 1] = dico[i];
7      sommet++;
8
9      dico[position].identif = malloc(strlen(identif) + 1);
10     if (dico[position].identif == NULL)
11         erreurFatale("Erreur interne (pas assez de mémoire)");
12     strcpy(dico[position].identif, identif);
13     dico[position].type = type;
14     dico[position].adresse = adresse;
15     dico[position].complement = complt;
16 }
```

Tableau trié et recherche dichotomique

```
1   ...
2   int placement;
3   ...
4   if (existe(lexeme, base, sommet - 1, &placement))
5       erreurFatale("identificateur déjà déclaré");
6   else {
7       ici se place l'obtention des informations type, adresse, complement, etc.
8       ajouterEntree(placement, lexeme, type, adresse, complement);
9   }
10  ...
```

Tableau trié et recherche dichotomique

- L'utilisation de tableaux triés permet d'optimiser la recherche, dont la complexité passe de O(n) à O(log2 n)
- Mais pénalise les insertions, dont la complexité devient O(n), puisqu'à chaque insertion il faut pousser d'un cran la moitié (en moyenne) des éléments du dictionnaire.
- Or, pendant la compilation d'un programme il y a beaucoup d'insertions et on ne peut pas négliger a priori le poids des insertions dans le calcul du coût de la gestion des identificateurs.

Arbre binaire de recherche

- **Rappel :** Analyse lexical et définition des mot réservés
- La gestion d'une table de mots réservés.
- En effet, le compilateur, ou plus précisément l'analyseur lexical, fait de nombreuses recherches dans cette table qui ne subit jamais la moindre insertion.

Arbre binaire de recherche

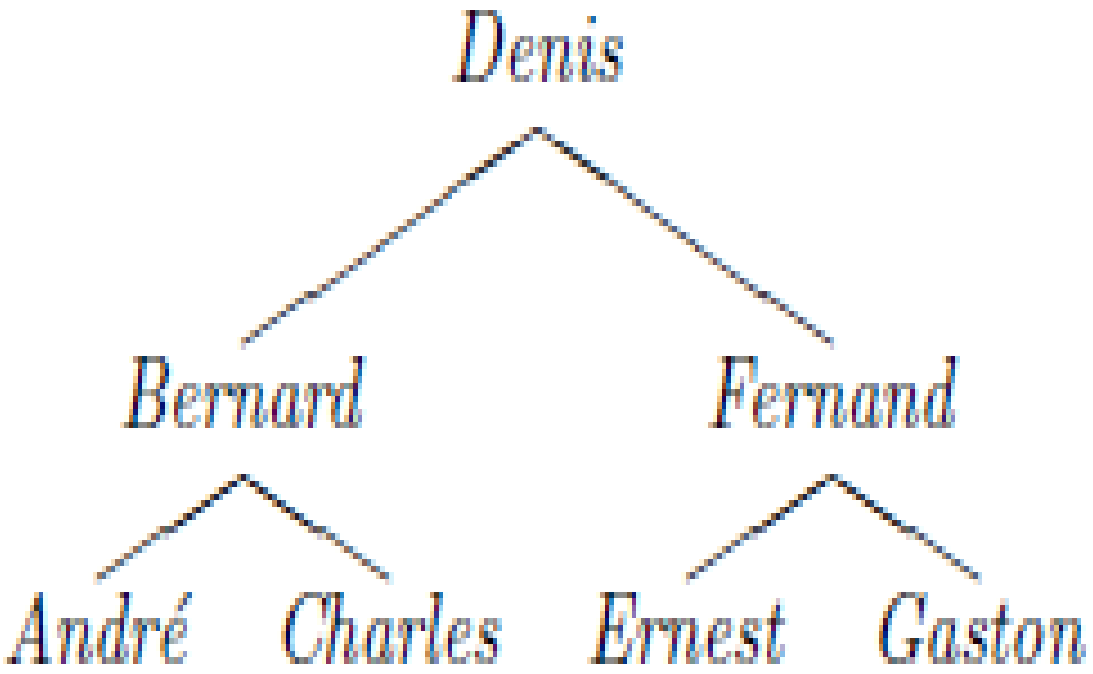
- Un ***arbre binaire de recherche***, ou ***ABR***, est un arbre binaire étiqueté par des valeurs appartenant à un ensemble ordonné, vérifiant la propriété suivante : pour tout nœud *p* de l'arbre:
  - pour tout nœud *q* appartenant au sous-arbre gauche de *p* on a

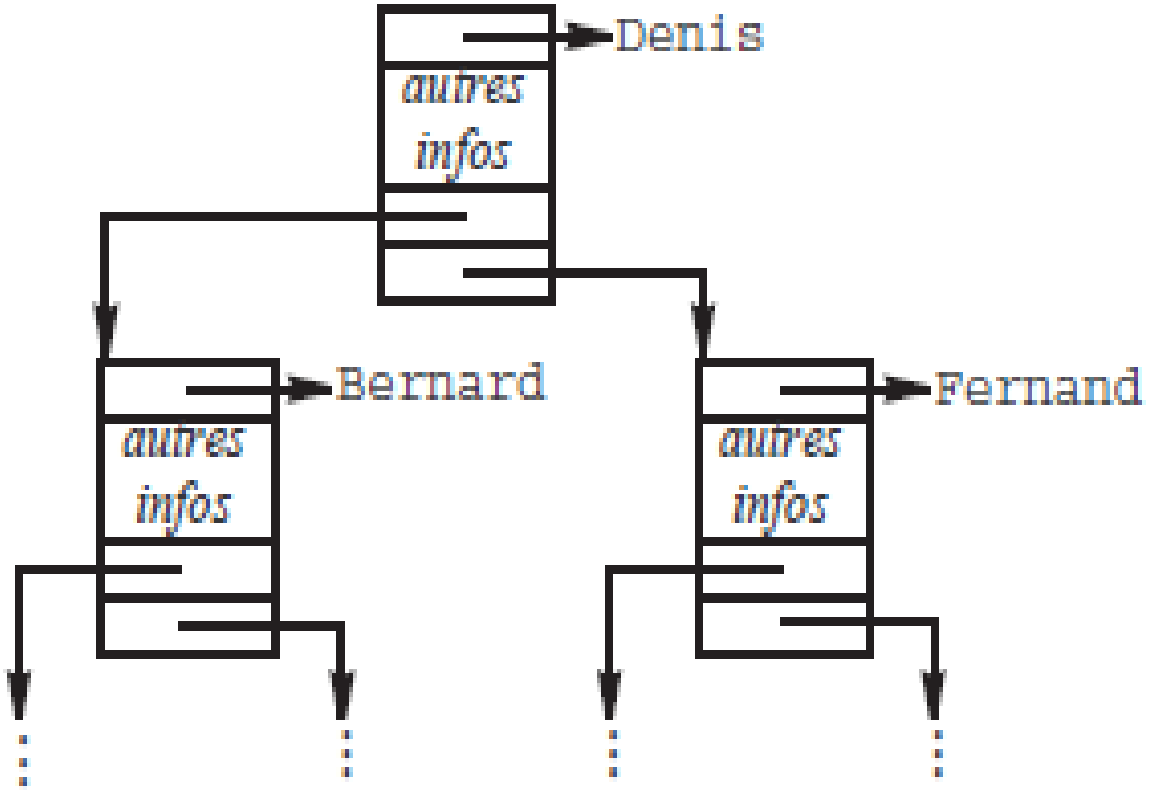
    $q \rightarrow info \leq p \rightarrow info$,
  - pour tout nœud *r* appartenant au sous-arbre droit de *p* on a

    $r \rightarrow info \geq p \rightarrow info$.

Arbre binaire de recherche

- Voici, par exemple, l'ABR obtenu avec les identificateurs Denis, Fernand, Bernard, André, Gaston, Ernest et Charles, ajoutés à l'arbre successivement et dans cet ordre :





Arbre binaire de recherche

```
1  typedef
2  struct noeud {
3      ENTREE_DICO info;
4      struct noeud *gauche, *droite;
5  } NOEUD;
6
7  NOEUD *dicoGlobal = NULL, *dicoLocal = NULL;
8
```

Arbre binaire de recherche

- Voici la fonction qui recherche le nœud correspondant à un identificateur donné.

```
2   NOEUD *rechercher(char *identif, NOEUD *ptr) {
3       while (ptr != NULL) {
4           int c = strcmp(identif, ptr->info.identif);
5           if (c == 0)
6               return ptr;
7           else
8               if (c < 0)
9                   ptr = ptr->gauche;
10              else
11                  ptr = ptr->droite;
12      }
13      return NULL;
14  }
```

Elle rend l'**adresse** du nœud cherché, ou **NULL** en cas d'échec de la recherche

Arbre binaire de recherche

- L'appel de la fonction **rechercher()**

```
17  ...
18  p = rechercher(lexeme, dicoLocal);
19  if (p == NULL) {
20      p = recherche(lexeme, dicoGlobal);
21      if (p == NULL)
22          erreurFatale("identificateur non déclaré");
23  }
24  exploitation des informations du nœuds pointé par p
25  ...
```

Arbre binaire de recherche

- Pendant la compilation des déclarations, les recherches se font avec la fonction insertion(),
- Cette fonction effectue la recherche et l'ajout d'un nouveau nœud.
- La rencontre d'un nœud associé à l'identificateur qu'on cherche est considérée comme une erreur grave.

- La fonction rend l'adresse du nœud nouvelle créé :

```
26
27  NOEUD *insertion(NOEUD **adrDico, char *identif, int type, int adresse, int complt) {
28      NOEUD *ptr;
29      if (*adrDico == NULL)
30          return *adrDico = nouveauNoeud(identif, type, adresse, complt);
31
32      ptr = *adrDico;
33      for (;;) {
34          int c = strcmp(identif, ptr->info.identif);
35          if (c == 0)
36              erreurFatale("identificateur deja déclaré");
37          if (c < 0)
38              if (ptr->gauche != NULL)
39                  ptr = ptr->gauche;
40              else
41                  return ptr->gauche = nouveauNoeud(identif, type, adresse, complt);
42          else
43              if (ptr->droite != NULL)
44                  ptr = ptr->droite;
45              else
46                  return ptr->droite = nouveauNoeud(identif, type, adresse, complt);
47      }
48  }
49
```

- Exemple d'appel (cas des déclarations locales)

```
50  ...
51  p = insertion( &dicoLocal, lexeme, leType, lAdresse, leComplement);
52  ...
```

- **NB**
  - Dans la fonction insertion, le pointeur de la racine du dictionnaire dans lequel il faut faire l'insertion est passé par adresse, c'est pourquoi il y a deux ** dans la déclaration NOEUD **adrDico.
  - Cela ne sert qu'à couvrir le cas de la première insertion, lorsque le dictionnaire est vide : le pointeur pointé par adrDico (en pratique il s'agit soit de dicoLocal, soit de dicoGlobal) vaut NULL et doit changer de valeur.

Arbre binaire de recherche

- RESTITUTION DE L'ESPACE ALLOUÉ
- L'implémentation d'un dictionnaire par un *ABR* possède l'efficacité de la recherche dichotomique,
  - car à chaque comparaison on divise par deux la taille de l'ensemble susceptible de contenir l'élément cherché,
  - sans ses inconvénients, puisque le temps nécessaire pour faire une insertion dans un *ABR* est négligeable.
- Cette méthode a deux défauts
  - La recherche n'est dichotomique que si l'arbre est équilibré, ce qui ne peut être supposé que si les identificateurs sont très nombreux et uniformément répartis ;
  - La destruction d'un dictionnaire est une opération beaucoup plus complexe que dans les méthodes qui utilisent un tableau.

Arbre binaire de recherche

- La destruction d'un dictionnaire, en l'occurrence le dictionnaire local, doit se faire chaque fois que le compilateur sort d'une fonction.
- Cela peut se programmer de la manière suivante

```
2  void liberer(NOEUD *dico) {
3      if (dico != NULL) {
4          liberer(dico->gauche);
5          liberer(dico->droite);
6          free(dico);
7      }
8  }
```

- Pour rendre l'espace occupé par un ABR il faut le parcourir entièrement (alors que dans le cas d'un tableau la modification d'un index suffit)

Adressage dispersé

- ***Adressage dispersé***, ou ***hash-code***
  - Technique de gestion d'une table de symboles
  - Très utilisée dans les compilateurs réels.
- Principe :
  - Au lieu de rechercher la position de l'identificateur dans la table,
  - On obtient cette position par un calcul sur les caractères de l'identificateur ;
  - Si on s'en tient à des opérations simples, un calcul est certainement plus rapide qu'une recherche.
- Fonction $h : I \rightarrow [0,N[$
  - I l'ensemble des identificateurs existant,
  - N la taille de la table d'identificateurs.
- Fonction $h : I \rightarrow [0,N[$ qui serait :
  - Rapide, facile à calculer ;
  - Injective, c'est-à-dire qui à deux identificateurs distincts ferait correspondre deux valeurs distinctes.

Adressage dispersé

- On ne dispose généralement pas d'une telle fonction car l'ensemble *I* des identificateurs présents dans le programme n'est pas connu a priori.
- De plus, la taille de la table n'est souvent pas suffisante pour permettre l'injectivité (qui requiert $N \geq |I|$).
- On se contente donc d'une fonction h prenant, sur l'ensemble des identificateurs possibles, des valeurs uniformément réparties sur l'intervalle [0,N[. C'est-à-dire que h n'est pas injective.
- Mais :
  - si $N \geq |I|$, on espère que les couples d'identificateurs i_1, i_2 tels que $i_1 \neq i_2$ et $h(i_1) = h(i_2)$ (on appelle cela une collision) sont peu nombreux ;
  - si $N < |I|$, les collisions sont inévitables.
    - Dans ce cas on souhaite qu'elles soient également réparties
    - Pour chaque $j \in [0,N[$ le nombre de $i \in I$ tels que $h(i) = j$ est à peu près le même, c.-à-d. $|I|/N$.
    - Il est facile de voir pourquoi : h étant la fonction qui place les identificateurs dans la table, il s'agit d'éviter que ces derniers s'amoncellent à certains endroits de la table, tandis qu'à d'autres endroits cette dernière est peu remplie, voire présente des cases vides.

Adressage dispersé

- Il est difficile de dire ce qu'est une bonne fonction de hachage.
- La littérature spécialisée rapporte de nombreuses solutions, mais il n'y a probablement pas de solution universelle, car une fonction de hachage n'est bonne que par rapport à un ensemble d'identificateurs donné.
- Parmi les conseils qu'on trouve le plus souvent :
  - Prenez *N* premier (une des recettes les plus données, mais plus personne ne se donne la peine d'en rappeler la justification) ;
  - Utilisez des fonctions qui font intervenir *tous* les caractères des identificateurs ;
    - En effet, dans les programmes on rencontre souvent des grappes de noms, par exemple : *poids*, *poids1*, *poidsTotal*, *poids maxi*, etc. ;
    - Une fonction qui ne ferait intervenir que les cinq premiers caractères ne serait pas très bonne ici ;
  - Ecrire des fonctions qui donnent comme résultat de grandes valeurs ; lorsque ces valeurs sont ramenées à l'intervalle [0,*N*[,
    - Par exemple par une opération de modulo, les éventuels défauts (dissymétries, accumulations, etc.) de la fonction initiale ont tendance à disparaître.

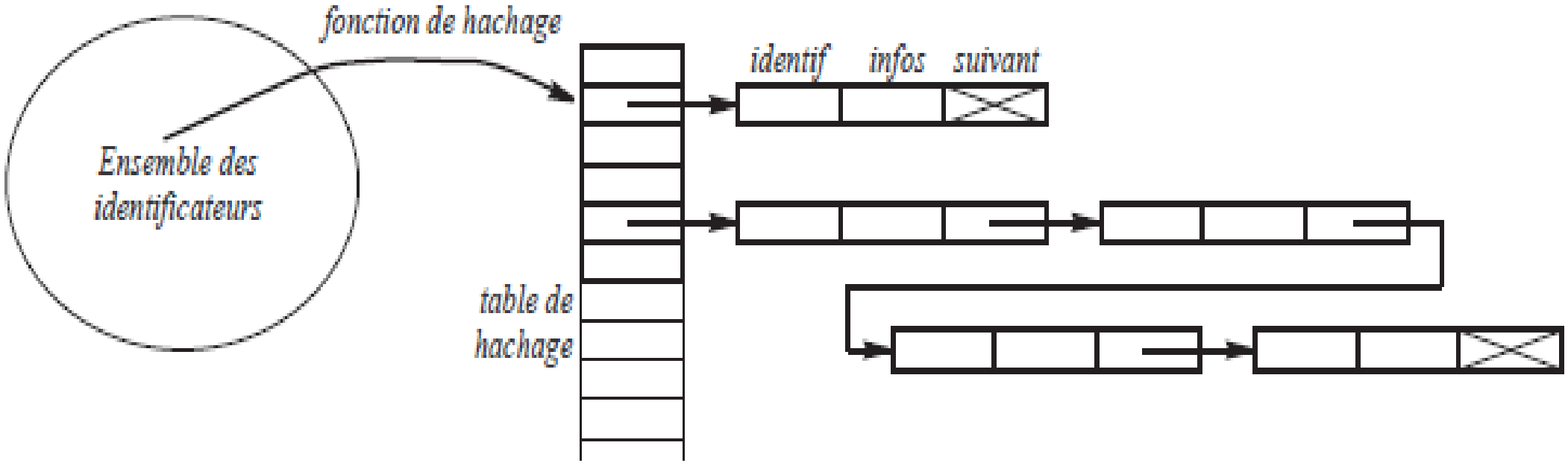
Adressage dispersé

- Une fonction assez souvent utilisée consiste à considérer les caractères d'un identificateur comme les coefficients d'un polynôme *P*(*X*).
- On calcule pour *P(X)* la valeur pour un *X* donné (ou, ce qui revient au même, à voir un identificateur comme l'écriture d'un nombre dans une certaine base).
- En C, cela donne la fonction :

```
2  int hash(char *ident, int N) {
3      const int X = 23; /* why not? */
4      int r = 0;
5      while (*ident != '\0')
6          r = X * r + *(ident++);
7      return r % N;
8  }
9
```

Adressage dispersé

- Dans le cas de l'adressage dispersé *ouvert*, la table qu'on adresse à travers la fonction de hachage n'est pas une table d'identificateurs,
- Mais une table de listes chaînées dont les maillons portent des identificateurs
- Si on note T cette table, alors T_j est le pointeur de tête de la liste chaînée dans laquelle sont les identificateurs i tels que $h(i) = j$.



Adressage dispersé

- Voici la fonction de recherche :

```
2   typedef
3   struct maillon {
4       char *identif;
5       autres informations
6       struct maillon *suivant;
7   } MAILLON;
8
9   #define N 101
10  MAILLON *table[N];
11
12  MAILLON *recherche(char *identif) {
13      MAILLON *p;
14      for (p = table[hash(identif, N)]; p != NULL; p = p->suivant)
15      if (strcmp(identif, p->identif) == 0)
16      return p;
17      return NULL;
18  }
19
```

Adressage dispersé

- La fonction qui se charge de l'insertion d'un identificateur :

```
20  MAILLON *insertion(char *identif) {
21      int h = hash(identif, N);
22      return table[h] = nouveauMaillon(identif, table[h]);
23  }
```

- La fonction **nouveauMaillon** définie comme ceci:

```
25  MAILLON *nouveauMaillon(char *identif, MAILLON *suivant) {
26      MAILLON *r = malloc(sizeof(MAILLON));
27      if (r == NULL)
28          erreurFatale("Erreur interne (problème d'espace)");
29      r->identif = malloc(strlen(identif) + 1);
30      if (r->identif == NULL)
31          erreurFatale("Erreur interne (problème d'espace)");
32      strcpy(r->identif, identif);
33      r->suivant = suivant;
34      return r;
35  }
```