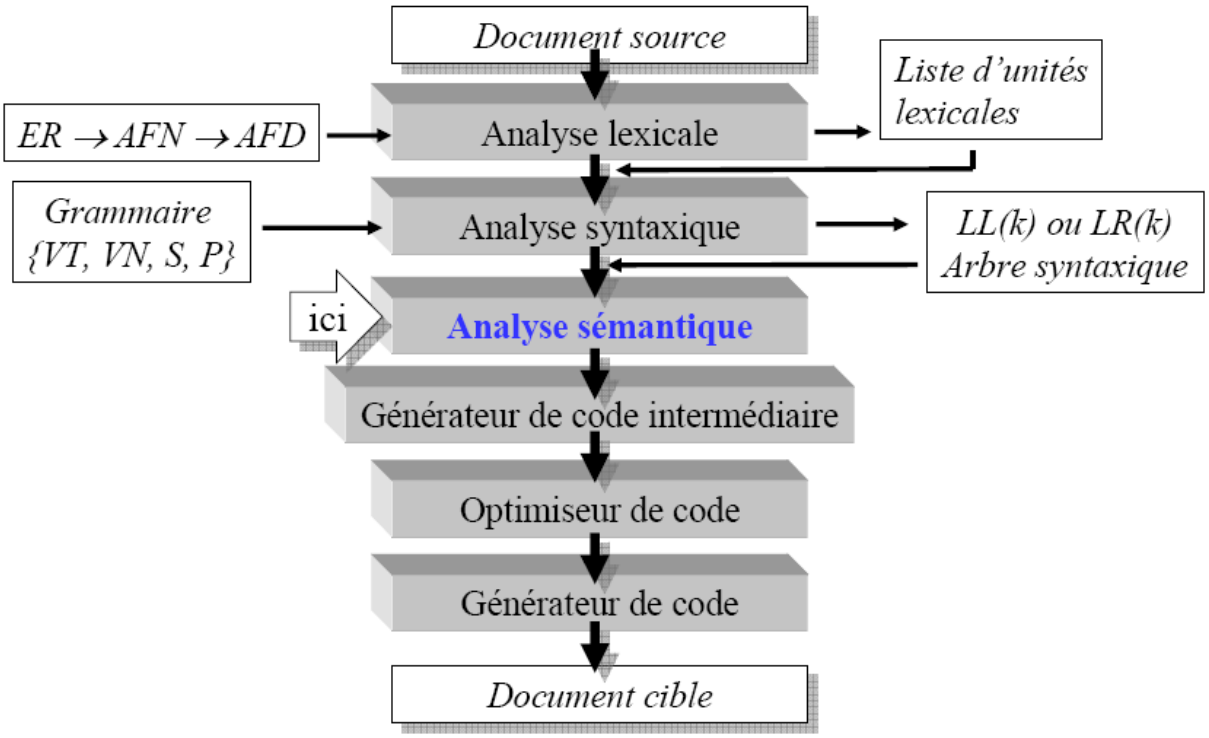
Analyse Sémantique

• Introduction :

Analyse Sémantique ?

- 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*.



- L'analyse sémantique vérifie que le programme satisfait bien un ensemble de règles de construction des programmes. Ces règles (appelées règles sémantiques) sont définis par un langage de programmation.

- Un programme lexicalement et syntaxiquement correct peut être sémantiquement faux.

- Les analyses lexicales et syntaxiques ne sont pas aptes à assurer la correction de l'usage des variables, des objets, des procédures…

Pour fixer les idées, l'ensemble des règles sémantiques qu'on va réaliser sont les suivantes :

- Construire la table des symboles.
- Contrainte 1 : Vérifier la redéfinition des variables déjà déclarées
- Contrainte 2 : Vérifier l'appel des procédures avec les bons arguments
- Contrainte 3 : Vérifier qu'une variable utilisée est bien déclarée.
- Contrainte 4 : Vérifier que les variables déclarées sont bien initialisées.
- Contrainte 5 : Vérifier qu'une variable déclarée est bien utilisée.

Ces règles doivent être implémentées dans un fichier « semantique.c ».

Ensuite, vous devez le déclarer dans le fichier yacc (définie dans la phase syntaxique) dans la partie déclarations :

```
%{

#include <stdio.h>

#include "semantique.c"

int yyerror(char const *msg);

int yylex(void);

extern int yylineno;

%}
```

Dans le même fichier yacc, vous allez rajouter les appels à vos fonctions implémentées dans le fichier « semantique.c » pour gérer les cinq contraintes.

Fichier « semantique.c » :

.1. Construction de la table des symboles :

- Dans les langages de programmation modernes, 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 associé à certains attributs, comme son type, sa classe…

⇨ Pour cela, on a crée une table des symboles qui est un tableau de structures dans laquelle chaque identifiant est associé à certains attributs bien déterminés:

Cette structure est définie par :

- Identif : le nom de l'identifiant
- Type : c'est le type de l'identifiant. Par défaut c'est réel.
- Classe : c'est une énumération qui indique si l'identifiant est une variable, nom de procédure ou un paramètre.
- Nb_param : le nombre de paramètres d'une procédure si l'identifiant est un nom de procédure.
- Is_used : champs qui indique si l'identifiant est bien utilisé.
- Is_init : champs qui indique si l'identifiant est bien initialisé.

Le domaine des définitions des différents attributs est le suivant :

Identif = {ID}	**Nb_param = {entier}**
Type = {entier}	**Is_used = {0,1}**
Classe = {variable, procédure, paramètre}	**Is_init = {0,1}**

Table global et table local :

- Un programme est essentiellement une collection de procédures entre lesquelles se trouvent des déclarations de variables. A l'intérieur des procédures se trouvent également des déclarations de variables.

On distingue deux types de variables :

- Les variables 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 locaux : Un objet local est visible dans la procédure où il est déclaré, depuis sa déclaration jusqu'µa la fin de cette procédure; il n'est pas visible depuis les autres procédure. En tout point où il est visible, un objet local *masque* tout éventuel objet global qui aurait le même nom.

En définitive, la table de symboles est divisée sur deux parties :

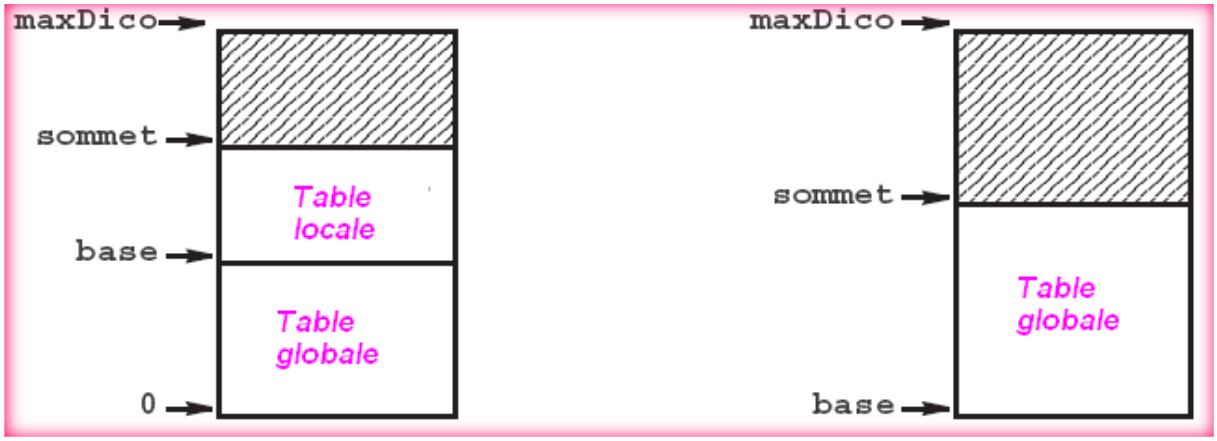
- Une *table globale*, contenant les noms des objets globaux couramment déclarés
- Une table *locale* dans lequel se trouvent les noms des objets locaux couramment déclarés (qui, parfois, masquent des objets dont les noms se trouvant dans le dictionnaire global).

Le fonctionnement de la table « dico » est la suivante :

- Lorsque le compilateur traite la déclaration d'un identificateur *i* en dehors de toute procédure, *i* est recherché dans la table globale, qui est la seul table existante en ce point ; normalement, il ne s'y trouve pas (sinon, erreur : identificateur déjà déclaré). Suite à cette déclaration, *i* est ajouté à la table globale.
- Lorsque le compilateur compile une instruction exécutable, forcément à l'intérieur d'une procédure, chaque identificateur *i* rencontré est recherché d'abord dans la table locale ; s'il ne s'y trouve pas, il est recherché ensuite dans la table global (si les deux recherches échouent, erreur : identificateur non déclaré). En procédant ainsi on assure le masquage des objets globaux par les objets locaux.

- Lorsque le compilateur quitte une procédure, la table local en cours d'utilisation est détruite, puisque les objets locaux ne sont pas visibles à l'extérieur de la procédure. Une table locale nouvelle, vide, est crée lorsque le compilateur entre dans une procédure.

L'implémentation simple de la table des symboles est comme suit :



Trois variables sont essentielles dans la gestion de la table :

- **MaxDico :** est le nombre maximum d'entrées possibles.
- **Sommet :** est le nombre d'entrées valides dans la table ; on doit avoir :

  Sommet <=maxDico
- **Base :** est le premier élément de la table 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).

Pour plus d'informations sur la partie implémentation de la table des symboles, **merci de regarder le document PolyCompil.pdf section 4 page 37** comportant la définition de la table des symboles ainsi que les différentes fonctions pour gérer cette table.

.2. Exemple : 1ère Contrainte : Vérifier la redéfinition des variables déjà déclarées :

Le principe est de vérifier à chaque ajout d'une entrée l'existence de la variable dans la table « dico » en espérant de ne pas la trouver sinon on déclenche un message d'erreur ***« identificateur déjà déclaré »*** :

Une fonction « checkIdentifier » (par exemple) doit être implémentée pour gérer cette contrainte.

Cette fonction est à appeler dans le fichier yacc lors du traitement d'une déclaration :

```
liste_identificateurs : ID {checkIdentifier ($1,0,yylineno);yyerrok;}
|error {yyerror (" identificateur attendu dans la ligne :\n"); yyerrok;}
|liste_identificateurs virgule ID {checkIdentifier ($3,0,yylineno);yyerrok;}
|liste_identificateurs error ID {yyerror (" virgule attendu dans la ligne :\n");
yyerrok;}
|liste_identificateurs virgule error {yyerror (" identificateur attendu dans la ligne
:\n"); yyerrok;};
```

La fonction « checkIdentifier » est une fonction définie dans le fichier « semantique.c » permettant d'afficher le message d'erreur « identificateur déjà déclaré » et ayant la signature suivante :

void checkIdentifier (char identif, int classe, int nbligne)*

- *identif* : c'est l'ID récupéré avec $1, $3 ($1, $3 sont la position de l'ID)
- *classe* : c'est le type définie dans la structure de la table de symbole (exemple : classe 0 pour dire que c'est une variable, classe 1 pour dire que c'est la nom d'une procédure ...)
- *nbligne* : le numéro de la ligne