TD n° 3
Analyse sémantique

1. Le langage

On veut compiler un langage dont la grammaire est à spécifier. A titre d'exemple une grammaire est donnée ci-après. Les symboles terminaux y sont représentés par des mots en lettres majuscules ou des signes entre apostrophes ; les symboles non terminaux par des mots en minuscules. Comme souvent, le symbole de départ, l'axiome, est le membre gauche de la première production, c'est-à-dire `programme`.

```
programme -> { declarationVariable }
             { declarationTableau }
             { declarationFonction } '.'

declarationVariable -> ENTIER IDENTIF ';'

declarationTableau -> TABLEAU IDENTIF '[' NOMBRE ']' ';'

declarationFonction -> FONCTION IDENTIF '(' [ IDENTIF { ',' IDENTIF } ]
')'
                       { declarationVariable }
                       instructionBloc

instruction -> instructionAffect
               | instructionBloc
               | instructionSi
               | instructionTantque
               | instructionAppel
               | instructionRetour
               | instructionEcriture
               | instructionVide

instructionAffect -> IDENTIF [ '[' expression ']' ] '=' expression ';'

instructionBloc -> '{' { instruction } '}'

instructionSi -> SI expression ALORS instruction [ SINON instruction ]

instructionTantque -> TANTQUE expression FAIRE instruction

instructionAppel -> APPEL IDENTIF argumentsEffectifs ';'

argumentsEffectifs -> '(' [ expression { ',' expression } ] ')'

instructionRetour -> RETOUR expression ';'

instructionEcriture -> ECRIRE '(' expression ')' ';'

instructionVide -> ';'

expression -> conjonction { OU conjonction }

conjonction -> comparaison { ET comparaison }
```



```
comparaison -> expArith [ ( EGAL | DIFFERENT | '<' | INFEG ) expArith ]

expArith -> terme { ( '+' | '-' ) terme }

terme -> facteur { ( '*' | '/' ) facteur }

facteur -> '(' expression ')'
           | NOMBRE
           | IDENTIF [ '[' expression ']' | argumentsEffectifs ]
           | LIRE '(' ')'
```

2. Analyse sémantique

L'étape d'analyse sémantique consiste à ajouter des vérifications sémantiques à votre analyseur syntaxique. Pour fixer les idées, disons que votre analyseur doit devenir capable de détecter des erreurs telles que :

- « *identificateur déjà déclaré* », dans une déclaration globale ou locale, si l'identificateur en question a déjà été déclaré au même niveau,
- « *identificateur non déclaré* », dans une instruction (i.e. un texte qui n'est pas une déclaration), si l'identificateur en question n'a pas été déclaré ni localement ni globalement,
- « *ce n'est pas une variable* », « *ce n'est pas un tableau* », sur un identificateur qui n'a pas la bonne classe, apparaissant dans une expression ou une instruction d'affectation,
- « *ce n'est pas une fonction* », « *mauvais nombre d'arguments* », lors d'un appel de fonction incorrect,

Pour être en mesure de faire de telles vérifications vous devez ajouter à votre analyseur une structure de données appelée ici *dictionnaire des identificateurs*. Pour cela, vous devez choisir :

- **La structure de chaque article du dictionnaire.** On vous conseille de définir une structure à quatre ou cinq champs : l'*identificateur* en question et un ensemble d'informations associées (des entiers) : la *classe*, l'*adresse* et un éventuel *complément* qui, en réalité, ne sera utile que pour les fonctions.

Pour le champ *identificateur* vous pouvez choisir soit le type tableau (avec une taille fixée) de caractères, soit le type pointeur de caractère. L'inconvénient dans le premier cas est que cela introduit une limitation du nombre de caractères des identificateurs et qu'en conséquence il va falloir modifier (légèrement) l'analyseur lexical. L'inconvénient dans le second cas est que les ajouts d'identificateurs au dictionnaire impliqueront des allocations dynamiques (*malloc*) et qu'il faudra donc ne pas oublier les libérations (*free*) correspondantes lors de la destruction d'un dictionnaire. Dans un premier temps, nous conseillons la première manière de faire.

Notre langage *L* est très simple du point de vue des types et dans notre compilateur la *classe* d'un identificateur est classique (`par exemple : C_FONCTION`, `C_ENTIER`, `C_TABLEAU`, etc.).

Les valeurs de ces champs sont des constantes définies à cet effet (n'utilisez pas les constantes définissant les unités lexicales, elles n'ont rien à voir avec ce qui nous occupe ici).

La question des *adresses* sera vue plus tard.

- **La structure du dictionnaire lui-même**. Si votre compilateur devait faire une carrière industrielle cette question serait très importante, car tous les compilateurs passent beaucoup de temps à rechercher des identificateurs dans le dictionnaire et il est primordial que ce dernier permette des recherches rapides. Mais, puisque dans un premier temps nous ne travaillons pas dans cet esprit, nous nous contenterons d'un dictionnaire dont les principales qualités sont la clarté et la simplicité et nous supporterons que les recherches n'y soient pas aussi rapides qu'elles le pourraient.

- **Dictionnaire global et local**. Lorsque le compilateur examine l'intérieur d'une fonction, deux dictionnaires coexistent : le dictionnaire local contient les identificateurs locaux à la fonction, le dictionnaire global contient les autres identificateurs. Lorsque le compilateur examine les éléments extérieurs aux fonctions, seul le dictionnaire global existe.

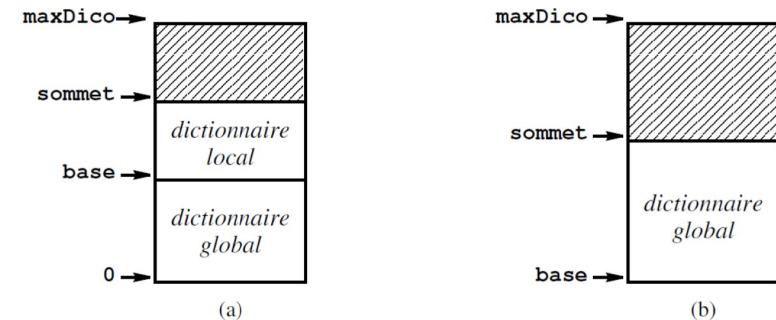
Le dictionnaire global est créé au début de la compilation, il existe jusqu'à la fin de celle-ci, il ne fait que grandir. Un dictionnaire local est créé, vide, chaque fois que commence la compilation d'une fonction. Ce dictionnaire existe durant la compilation de la fonction, et est détruit lorsque cette compilation se termine.

La recherche d'un identificateur dans le(s) dictionnaire(s) se produit dans trois circonstances différentes :

- lorsque le compilateur traite la partie exécutable (c.-à-d les instructions) d'une fonction. La recherche se fait alors d'abord dans le dictionnaire local, ensuite dans le dictionnaire global. En principe, une telle recherche doit aboutir, son échec déclenche une erreur « Identificateur non déclaré »;
- lorsque le compilateur traite la partie déclarative d'une fonction. La recherche se fait alors uniquement dans le dictionnaire local. En principe, une telle recherche doit échouer, son succès déclenche une erreur « Identificateur déjà déclaré dans cette fonction » ;
- lorsque le compilateur se trouve en dehors de toute fonction (il est donc certainement en train de compiler une déclaration globale). La recherche se fait alors dans le dictionnaire global, qui est le seul à exister à ce moment. Une telle recherche doit échouer, son succès déclenche une erreur « Identificateur déjà déclaré ».

3. Implémentation

Une implémentation, simple mais suffisante, du dictionnaire : un tableau de couples (identificateur, attributs) contrôlé par deux indices base et sommet comme le montre la figure.



Les ajouts dans le dictionnaire se font toujours dans la case d'indice sommet, en incrémentant cet indice. Les recherches se font :

- durant la compilation d'une partie exécutable, en parcourant à reculons les indices [0, sommet[,
- dans une partie déclarative, en parcourant les indices [base, sommet[.

Avec cette implémentation des dictionnaires, les opérations de création d'un dictionnaire local (lorsque le compilateur « rentre » dans une fonction) et de destruction du dictionnaire local (lorsque le compilateur « sort » de la fonction) se réduisent à des opérations élémentaires sur les indices base et sommet :

- entrée dans une fonction : base ← sommet ;
- sortie d'une fonction : sommet ← base ; base ← 0 ;

3.1. L'interface du dictionnaire

Ce sont les fonctions et variables à travers lesquelles les autres parties de l'analyseur utilisent le dictionnaire. Sachez qu'on peut avoir tous les services nécessaires à travers le tableau `dico`, les indices `base` et `sommet` et deux fonctions simples :

```
int recherche(char *identificateur, int haut, int bas);
```

qui recherche dans le dictionnaire un article associé à l'`identificateur` indiqué. La recherche se fait *à reculons* de `haut` – 1 jusqu'à `bas` (il est supposé que `bas` <= `haut`; si `bas` = `haut` c'est que le dictionnaire est vide). La fonction renvoie l'indice du premier article convenant, ou une valeur négative si un tel article n'existe pas, et

```
int ajout(char *identificateur);
```

qui crée un nouvel article dans le dictionnaire (au sommet) et en renvoie l'indice. Le champs identificateur est initialisé avec l'`identificateur` indiqué (attention, s'agissant de chaînes de caractères, une simple affectation ne suffit pas...), les autres champs sont initialisés à zéro. En cas de débordement du dictionnaire, cette fonction se charge de déclencher une erreur qui termine la compilation.

3.2. Exemples d'utilisation

Exemple 1 : utilisation du dictionnaire à l'occasion d'une déclaration. Examinons ce que devient l'analyse d'une déclaration de variable, dans le cas de la grammaire donnée en exemple plus haut. La production est

```
declarationVariable -> ENTIER IDENTIF ';'
```

Dans l'analyseur syntaxique cela s'est traduit par une fonction

```
void declarVariable(void) {
    /* quand on vient ici il est certain que uc == ENTIER */
    uc = yylex();
    if (uc != IDENTIF)
        erreur("identificateur attendu");
    uc = yylex();
    if (uc != ';'')
        erreur("';' attendue");
    uc = yylex();
}
```

Après l'introduction du dictionnaire, cela devient

```
void declarVariable(void) {
    int id;
    /* quand on vient ici il est certain que uc == ENTIER */
    uc = yylex();
    if (uc != IDENTIF)
        erreur("identificateur attendu");
    id = recherche(lexeme, sommet, base);
    if (id >= 0)
        erreur("identificateur deja declaré");
    id = ajout(lexeme);
    dico[id].classe = contexteLocal ? C_VAR_LOCALE : C_VAR_GLOBALE;
    dico[id].type = T_ENTIER;

    uc = yylex();
    if (uc != ';')
        erreur("';' attendue");
    uc = yylex();
}
```

Cela permet d'utiliser la même fonction `declarVariable` pour analyser les déclarations de variables globales et les déclarations de variables locales.

Exemple 2 : utilisation du dictionnaire dans la partie exécutable. Regardons l'instruction « appel d'une fonction ». La grammaire dit

```
instructionAppel -> APPEL IDENTIF argumentsEffectifs ';'
```

Dans l'analyseur syntaxique cela se traduit par

```
void instructionAppel(void) {
    /* quand on vient ici il est certain que uc == APPEL */
    uc = yylex();
    if (uc != IDENTIF)
```



```
        erreur("identificateur attendu");
    uc = yylex();
    argumentsEffectifs();
    if (uc != ';')
        erreur("';' attendue");
    uc = yylex();
}
```

Voici ce qu'ajoute le dictionnaire :

```
void instructionAppel(void) {
    int id;
    /* quand on vient ici il est certain que uc == APPEL */
    uc = yylex();
    if (uc != IDENTIF)
        erreur("identificateur attendu");
    id = recherche(lexeme, sommet, 0);
    if (id < 0)
        erreur("identificateur non declare");
    if (dico[id].classe != C_FONCTION)
        erreur("ce n'est pas une fonction");
    uc = yylex();
    argumentsEffectifs(id);
    if (uc != ';')
        erreur("';' attendue");
    uc = yylex();
}
```

N.B. Ci-dessus on passe *id*, l'indice dans le dictionnaire, à *argumentsEffectifs* afin que cette dernière, chargée d'analyser les arguments effectifs de l'appel, puisse vérifier qu'il y en a le nombre voulu.

3.3. Les adresses des variables globales

Les adresses des variables globales sont relatives à une certaine *base* qui n'est pas connue durant la compilation, mais uniquement lors de l'exécution. Cette base est notée **BEG** (comme Base de l'Espace Global) et a pour valeur l'indice de la première cellule mémoire libre à la suite du code produit par le compilateur. Il est clair que la valeur de **BEG** n'est pas connue durant la compilation : il faut attendre la fin de celle-ci pour connaître la taille du code produit.

Il découle de tout cela que la première variable globale a l'adresse `0` (qui, à l'exécution, se traduira par `BEG + 0`), la deuxième variable globale l'adresse `1` (`BEG + 1`), la troisième l'adresse `2`, etc. La règle est simple : l'adresse relative à **BEG** d'une variable globale est la somme des tailles des variables que le compilateur a précédemment rencontrées.

Pour gérer cela il suffit donc de se donner un compteur, appelons-le `adresseGlobaleCourante`, initialisé à 0 et incrémenté de 1 à chaque déclaration de variable.

Les tableaux sont traités de la même manière, sauf que lors de la déclaration d'un tableau `adresseGlobaleCourante` n'est pas incrémenté de 1 mais de la taille du tableau.

4. Travail à faire

Vous devez écrire les définitions des types, variables et les deux fonctions, *recherche* et *ajout*, mentionnés plus haut, puis passer en revue votre analyseur pour déceler *tous* les endroits où il faut ajouter (il ne s'agit que d'ajouts!) du code nouveau pour effectuer l'analyse sémantique.

Pour tester le dictionnaire réalisé

- écrivez des codes sources qui provoquent les erreurs sémantiques possibles ;
- ajoutez à votre analyseur du code provisoire affichant le dictionnaire, par exemple à la fin de l'analyseur de chaque fonction (afin de contrôler le dictionnaire local correspondant).