Rapport Projet Mini-compilateur Pascal

Phase Analyse Sémantique

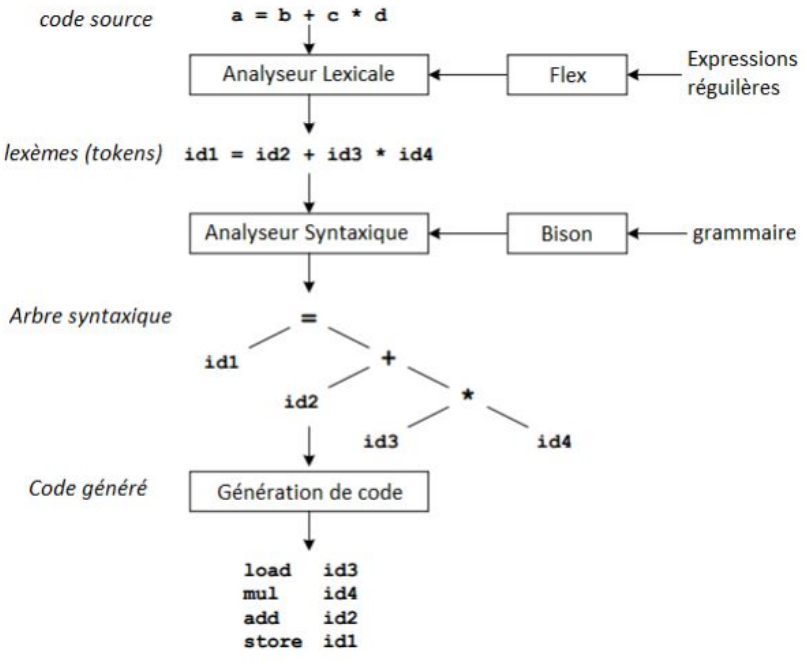
Cadre générale

Le but de ce travail est de programmer en langage C (ou C++) un compilateur de codes sources programmés à l'aide d'un sous-ensemble du langage Pascal appelé Mini-Pascal.

Le processus de compilation d'un programme consiste en un certain nombre de phases; en pratique les compilateurs sont écrits pour être capables de les réaliser ensemble, en faisant une seule passe dans sa donnée. Cependant, on va présenter ces phases séparément pour mieux comprendre le rôle de chacune de celles-ci.

Les quatres grandes phases:

1. L'analyse lexicale
2. L'analyse syntaxique
3. L'analyse sémantique
4. La production du code objet



Analyse Sémantique

La phase d'analyse sémantique est réalisée dans le fichier ".y" avec la définition des fonctions nécessaires dans le fichier "semantique.c".

Pour la construction de la table de symboles, on a défini une structure liste chaînée qui va contenir des noeuds, chaque noeud représente un symbole et contient son nom, type, classe , son nombre de paramètres s'il s'agit d'une procédure, s'il est initialisée et s'il est utilisée.

```
typedef enum {
    NODE_TYPE_UNKNOWN,
    tInt
} TYPE_IDENTIFIANT;

typedef enum {
    CLASSE_UNKNOWN,
    variable,
    procedure,
    parametre
} CLASSE;

struct NOEUD
{
    char* nom;
    TYPE_IDENTIFIANT type;
    CLASSE classe;
    int isInit;
    int isUsed;
    int nbParam;

    struct NOEUD * suivant;
};
typedef struct NOEUD * NOEUD;
typedef NOEUD TABLE_SEMANTIQUE;
```

On a défini les fonctions nécessaires à la gestion de ces structures :

```
NOEUD creerNoeud (const char* nom, TYPE_IDENTIFIANT type, CLASSE classe, NOEUD suivant);
NOEUD insererNoeud (NOEUD noeud, TABLE_SEMANTIQUE table);
NOEUD chercherNoeud (const char* nom, TABLE_SEMANTIQUE table);
```

Afin d'utiliser ces fonctions dans le fichier yacc on doit les inclure :

```
#include "semantique.c"
```

On peut maintenant utiliser ces fonctions dans la grammaire afin d'effectuer les actions souhaitées.

```
entete_methode          : PROCEDURE { g_IfProc = 1; }
                          IDENTIFIER
                        {
                            if( chercherNoeud(nom, table) ){
                                yyerror("Procedure already defined");
                            }else{
                                g_noeudProc = creerNoeud(nom, NODE_TYPE_UNKNOWN, procedure, NULL);
                                table = insererNoeud(g_noeudProc, table);
                            }
                            g_IfProcParameters = 1;
                        }
                          arguments
                        {
                            g_noeudProc->nbParam = g_nbParam;
                            g_nbParam = 0;
                        }
                        ;
```

On a déclaré une variable partagée entre l'analyseur lexical (fichier .lex) et l'analyseur syntaxique (fichier .yacc). Ainsi, on a pu à chaque fois récupérer la valeurs du non terminal lu et l'envoyer à la fonction adéquate pour définir les règles sémantiques.

```
// chaine de caractères partagée avec l'analyseur lexical
char nom[256];
```

```
liste_identificateurs   : IDENTIFIER {
                              checkIdentifier(nom,nbline);
                          } ',' liste_identificateurs
                          | IDENTIFIER {
                              checkIdentifier(nom,nbline);
                          }
                          ;
```

Les fonctions définies dans le fichier "semantique.c" pour assurer les contraintes sémantiques du langage.

```
void checkIdentifier(char* nom, int nbline);
int checkIdentifierDeclared(char* nom, int nbline);
void varInitialized (char* nom);
void checkVarInit(char * nom, int nbline);
void endProc(int nbline);
void destructSymbolsTable( TABLE_SEMANTIQUE SymbolsTable );
int print_error(char* msg, int nbline);
```