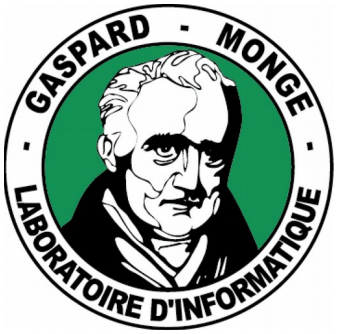
Guy LUONG
Karim HAMDI

Projet Compilation
Documentation



Licence Informatique
Université Paris-Est Marne-la-Vallée

Introduction :

Ce projet consiste à créer un compilateur d'un langage nommé *tpc*, qui est un sous-ensemble du langage *C*, qui traduit du *tpc* en code machine virtuelle *MLV* à l'aide des logiciels *flex* pour l'analyse lexicale, et *bison* pour l'analyse syntaxique.

La grammaire de base étant donnée, il fallait écrire le fichier *.lex* pour le logiciel *flex*, et compléter cette grammaire en y mettant les priorités, les types, etc. Y compris les actions, qui seront principalement de la traduction du code *tpc* en code machine virtuelle *MLV* (documentation disponible dans *docVM.pdf*). Nous allons ici détailler la composition de la table des symboles, tout ce qu'il est possible de faire, et tous les problèmes à la fin.

Table des symboles :

Nous avons fait deux tables de symboles :

- ```
  typedef struct {
      char name[IDENT_MAX]; /* Identificateur sur 64 caractères */
      int type; /* Entier / Constante / Caractère */
      int size; /* Taille de la variable, 1 pour tout ce qui n'est pas tableau, la taille du tableau sinon */
      int position; /* Position dans la pile lors de la création de la variable */
  }var_sym;
  ```

- ```
  typedef struct {
      char name[IDENT_MAX]; /* Identificateur sur 64 caractères */
      int return_type; /* Type de retour de la fonction */
      int nb_arg; /* Nombre d'arguments */
      int nb_vars_max; /* Taille maximale du tableau de fonction (alloué et réalloué sinon) */
      int num_lab; /* Numéro du label pour créer le saut vers la fonction */
      int nb_vars; /* Nombre de variables actuellement dans le tableau */
      var_sym *fun_vars; /* Tableau de variables locales */
  }fun_sym;
  ```

Et nous avons créé un tableau global de fonctions.

- *fun_sym* funs = NULL;*

Le tableau est global car nous voulons y avoir accès dans toute

l'analyse du code en *tpc*.
A l'indice 0 du tableau, on y range la fonction virtuelle (de label 1), qui comprend toutes les constantes et les variables globales.

Pour gérer les fonctions, notamment le label indiquant le début de fonction et son saut, on utilise une variable globale nommée *current_label*, qui donne le label de la fonction en cours d'analyse. En effet, ce langage ne permet pas la déclaration de fonction dans une fonction, et est ordonné. On y met d'abord les constantes globales, puis les variables globales, puis les fonctions. On peut donc facilement associer les variables à leur fonction.

Après avoir déclaré les variables globales et les constantes, on met un appel à *main*, qui doit être présente dans le programme, comme en C, c'est là où le programme commence à exécuter les instructions.

Ce qu'il est possible de faire :

Avec ce programme, il est possible de :
- calculer des expressions binaires (addition, multiplication, comparaison,...), unaires (négation, opposé,...), ou même des appels de fonction
- afficher des valeurs à l'écran à l'aide de *print* (), suivant le type de l'expression donnée pour *print*, *print* affiche soit un caractère, soit un entier.
- lire des entiers avec *read* () ou caractères avec *readch* (),
- déclarer des variables de types constante / entier / caractère,
- les modifier (il n'est pas possible de les initialiser à la déclaration)
- déclarer des fonctions de types de retour entier / caractère, ou sans type de retour, à paramètres passés par valeur.
- appeler ces fonctions,
- faire des fonctions récursives.
- faire des structures conditionnelles (if, if/else, if ternaire style Python),
- faire des boucles (while)
- utiliser des variables globales/constantes depuis une fonction.

Pour évaluer une expression booléenne (pour le et/ou logique), on utilise l'évaluation paresseuse : si un des membres de l'expression booléenne suffit à déterminer le résultat, on ne calculera pas le reste. On teste les membres de gauche à droite.

Problèmes :

- L'implantation de tableaux nécessite une refonte du code (dans la grammaire, pour la règle commençant par LValue). Avec des modifications simples, il était possible de lire un tableau en le parcourant, mais il n'était pas possible d'écrire dans toutes les cases du tableau. On ne pouvait écrire que dans la première case. Comme on ne pouvait pas écrire dans toutes les cases du tableau, nous avons retiré cette fonctionnalité.

- Il n'y a pas de vérification pour savoir si dans tous les cas de la fonction, il y a un return. Cela peut poser des problèmes à l'exécution. Il ne faut pas oublier le return ! Les types du retour ne sont pas vérifiés.

- Il y a un bogue (syntax error) si la première fonction qu'on déclare n'est pas de type *void*. Il y a conflit avec une déclaration de variable : les deux commencent par TYPE IDENT, mais une action est exécutée dans le cas de déclaration de variable, faisant en sorte que cette fonction soit considérée comme une variable. La parenthèse suivant cette action provoque la *syntax error*. Le type *void* est une spécificité des fonctions, une fois qu'on a le type void, on sait que si on rencontre un TYPE IDENT hors de fonctions, il s'agira forcément d'une fonction. De là, on pourra écrire des fonctions par exemple de type *entier* qui ne provoqueront pas de *syntax error.*

Exemples ne marchant pas :

-
```
entier factorielle (entier x) {
        if (x <= 1) {
                return 1;
        }
        return x * factorielle (x - 1);
} /* 1e fonction de type non void */

void main (void) {
        print (factorielle (5));
        return ;
}
```

-
```
void print_double (entier a) {
        print (a*2) ;
```

```
} /* Il manque le return */

void main (void) {
        print (factorielle (5));
        return ;
}
```

Exemple de code qui marche :

```
void a (void) {
        print (5) ;
        return ;
}

entier factorielle (entier x) {
        if (x <= 1) {
                return 1;
        }
        return x * factorielle (x - 1);
}


void main (void) {
        print (factorielle (5));
        return ;
}
```