Université Gustave Eiffel

BERNIER Aurélien
BLONDEL Lucas

Compilateur : Manuel utilisateur

Sommaire

I - Préambule

Présentation

L'objectif du projet était de réaliser un compilateur d'un langage donné dérivé du C à partir du projet d'analyse syntaxique réalisé au plus tôt dans l'année.

Fonctionnement

La compilation d'un programme se déroule en plusieurs étapes :

- Le programme va d'abord être analysé lexicalement à l'aide de flex puis syntaxiquement à l'aide de bison. Toute erreur lexicale ou syntaxique détectée sera signalée et le programme ne sera pas compilé. Si aucune erreur n'est signalée, un arbre abstrait va être construit à partir du programme TPC.

- Plusieurs parcours de l'arbre vont ensuite être réalisés afin de repérer les éventuelles erreurs sémantiques. Un 1er parcours de l'arbre abstrait qui va permettre de construire les tables de symboles et signaler les erreurs sémantiques de redéfinition (variable déjà déclarée ou fonction redéfinie). Ici, on utilise le module table.h.

- Un 2ème parcours permet de vérifier les autres erreurs sémantiques comme celles de type, main non déclaré ou variables non déclarées. Ici, on utilise le module sem.h.

- Un dernier parcours va s'occuper de la traduction de l'arbre en programme nasm. Ici, on utilise le module trad.h.

II - Le langage TPC

Identificateurs

Tout identificateur utilisé comme variable ou paramètre dans un programme doit être déclaré avant son utilisation et dans la partie de déclaration appropriée.

Une variable globale et une fonction ne peuvent pas avoir le même nom.

Une variable locale et un paramètre d'une même fonction ne peuvent pas avoir le même nom.

Fonctions

Tout programme doit comporter la fonction particulière main par laquelle commence l'exécution et qui renvoie obligatoirement un int.

L'utilisation des instructions return doit être conforme au type de retour de la fonction dans laquelle elles apparaissent. Un appel de fonction ne peut pas être utilisé comme expression si le type de retour de la fonction appelée est void.

Les arguments d'une fonction sont transmis par valeur, les pointeurs n'étant pas possibles en TPC. Les fonctions récursives directes et indirectes sont autorisées. Toute déclaration de fonction sans paramètre doit comporter « void ».

Des fonctions pré-écrites sont disponibles : getchar(), getint(), putint(ENTIER), putchar(CARACTERE)

Les fonctions getchar() et getint() permettent de lire un caractère et un entier en notation décimale saisis au clavier par l'utilisateur. Elles renvoient le caractère ou l'entier comme valeur de retour. Si le premier caractère lu n'est pas un chiffre ni un signe moins, getint() termine l'exécution du programme et le programme renvoie la valeur de retour 5, qui code une erreur d'entrées-sorties.

Les fonctions putchar() et putint() permettent d'afficher sur la sortie standard un caractère et un entier passés en paramètre. L'entier est affiché en notation décimale. Les fonctions putchar() et putint() ne renvoient pas de valeur.

Le compilateur écrit ces fonctions directement en nasm, et le code tpc ne pourra pas les redéfinir.

Types

Le typage des expressions est à peu près comme en C :

- Le type int représente les entiers signés codés sur 4 octets.

- Toute expression de type char à laquelle on applique une opération est implicitement convertie en int. Par exemple, si count est de type int, count='a' est correct et ne doit pas provoquer d'avertissement.

- Toute expression peut être interprétée comme booléenne, avec la convention que 0 représente "faux" et tout autre valeur "vrai", et le résultat de toute opération booléenne est de type int. En particulier, l'opérateur de négation produit comme résultat 1 quand on l'applique à 0.

- Si on affecte une expression de type int à une LValue de type char, le compilateur émet un avertissement (warning) mais produit un programme cible fonctionnel. Par exemple, si my_char est de type char, my_char=97 doit provoquer un avertissement.

- Dans les appels de fonctions, on considère que chaque paramètre effectif correspond à une affectation pour ce qui concerne le typage ; l'instruction return Exp est aussi considérée comme une affectation. Par exemple, dans une fonction qui renvoie un int, return 'a' est correct, mais dans une fonction qui renvoie un char, return 97 doit provoquer un avertissement.

Tableaux

La sémantique pour les tableaux est celle du langage C, sauf indication contraire ci-dessous.

Dans une déclaration de variable de type tableau, la taille du tableau devra être strictement positive. Un tableau de taille nulle est une erreur sémantique.

Dans une instruction, les seuls opérateurs applicables à un tableau sont les crochets, comme dans duration[2], et le fait de passer le tableau en argument dans un appel de fonction, comme dans bissextile(duration,12);.

Appliquer un autre opérateur à un tableau, ou l'utiliser comme booléen, comme dans if (duration) return;, est une erreur sémantique.

III - Utilisation du compilateur

Compilation et lancement

Pour compiler le compilateur, il faut utiliser la commande : make

Pour utiliser le compilateur, il faut utiliser : ./bin/tpcc [*OPTIONS*] < *[FICHIER*]
Les options possibles sont :
-t / --tree pour afficher l'arbre syntaxique
-h / --help qui affiche un court mode d'emploi du programme
-s /--symtabs qui affiche les tables de symboles

Un jeu de test est disponible pour tester le compilateur ou pour servir d'exemple de programme écrit en TPC.

Pour le lancer, il suffit d'utiliser la commande : bash test/test.bash, un rapport de tests va être écrit et placé dans le répertoire « test » donnat une note au compilateur.

Exemple d'utilisation

Utilisons le compilateur sur un programme TPC : ./bin/tpcc -s < test.tpc.

```
int tab[5], a;

int sum(int a, int b) {
    return a + b;
}

int main(void) {
    int b;
    b = 4;
    a = 2;
    putint(getint());
    return sum(a, b);
}
```

Figure 1: Programme écrit en TPC

Nous obtenons alors dans le terminal les tables de symboles suivantes :

```
Table 0 :
|Type : int array, Ident : tab, Taille : 5, Fonct : non, Adr : 0
|Type : int, Ident : a, Taille : 0, Fonct : non, Adr : 40
|Type : void, Ident : putint, Taille : 0, Fonct : oui, Adr : 48
|Type : int, Ident : getint, Taille : 0, Fonct : oui, Adr : 56
|Type : int, Ident : sum, Taille : 0, Fonct : oui, Adr : 64
|Type : int, Ident : main, Taille : 0, Fonct : oui, Adr : 72

Table 1 :
|Type : int, Ident : n, Taille : 0, Param : oui, Adr : -8

Table 2 :
VIDE

Table 3 :
|Type : int, Ident : a, Taille : 0, Param : oui, Adr : -8
|Type : int, Ident : b, Taille : 0, Param : oui, Adr : -16

Table 4 :
|Type : int, Ident : b, Taille : 0, Param : non, Adr : -8
```

Figure 2: Tables produites à partir du programme TPC figure 1

Nous pouvons noter plusieurs éléments :

- pour simplifier, nous avons décidé par convention que la table 0 serait celle des variables globales et que les tables suivantes seraient celles des fonctions.

- le type d'une variable ou fonction et représenté par un entier de 0 à 4. 0 correspond à void, 1 à char, 2 à int, 3 à char array et 4 à int array. Nous avons ajouté 2 types différents pour les tableau pour simplifier la vérification de type plus tard.

- une table de fonction sans paramètre ni variable locale est simplement noté VIDE (ici getint)

- le champ Taille correspond à la taille d'un tableau, il va donc valoir 0 dans la cas d'une variable

- dans les tables de fonction, le champ param indique si la variable est un paramètre. De même dans la table globale le champ fonct indique si il s'agit d'une fonction.

- adr correspond à l'adresse relative de la variable dans la table de la fonction. Dans la table globale il s'agit d'une adresse globale. Afin de simplifier la traduction en nasm, les adresses des variables dans les tables de fonctions sont négatives car les adresses de variables locales sont négatives par rapport à rbp.

- par convention, les fonctions sont toujours après les variables globales dans la table globale et les paramètres sont toujours avant les variables locales dans les tables des fonctions.

- les tables des fonctions sont dans le même ordre que les déclarations de fonctions.

Après analyse lexicale, syntaxique et sémantique, le programme est traduit en nasm dans un fichier nommé _anonymous.asm placé à la racine du projet.

Il est ainsi possible de le compiler via la commande make bin/_anonymous et de l'exécuter avec ./bin/_anonymous.