OUMAKHLOUF Selym
LUSTGARTEN Léo

Projet de Compilation

Licence d'Informatique

– 2023-2024 –

24/05/2024

Introduction

● Objectif du projet

Le projet de compilation a pour objectif de développer un compilateur en langage C pour un langage source nommé tpc, qui est un sous-ensemble simplifié du langage C.
Le compilateur doit traduire le code tpc en assembleur nasm, respectant la syntaxe Unix et les conventions d'appel AMD 64.
Ce projet fait suite au travail effectué lors du semestre précédent où nous avions développé un analyseur syntaxique en utilisant flex et bison et vise à approfondir nos compétences en conception et implémentation de compilateurs, avec un accent particulier sur les différentes phases de la compilation :

- ☐ l'analyse syntaxique,
- ☐ la génération de code et la gestion d'erreurs.

● Consignes et Contraintes

Le compilateur doit vérifier les erreurs lexicales et syntaxiques mais aussi détecter les erreurs sémantiques. La fonction `main` doit être présente dans chaque programme et doit renvoyer un entier. De plus, certaines fonctions spécifiques (`getchar()`, `getint()`, `putint()`, `putchar()`) doivent être implémentées directement en assembleur. Les tableaux doivent être gérés conformément aux règles définies, et des messages d'erreur clairs doivent être fournis en cas de problème.

Organisation du Projet

• Structure des Répertoires

Pour assurer une organisation claire et structurée du projet, nous avons respecté les consignes données et avons organisé notre dépôt de la manière suivante :

- ☐ `src` contient tous les fichiers sources que nous avons écrits. Il comprend notamment les fichiers de code C, les fichiers de définition de grammaire pour flex et bison ainsi que d'autres fichiers nécessaires au bon fonctionnement du compilateur.
  - ☐ `exit_functions.c` : fonctions à utiliser avec on_exit()
  - ☐ `symbol_table.c` : gère la table des symboles et les erreurs
  - ☐ `tpcc.lex` : définition lexicale pour flex
  - ☐ `tpcc.y` : grammaire syntaxique (bison) et contient le main
  - ☐ `traducteur.c` : génération du code assembleur
  - ☐ `tree.c` : fonction pour l'arbre de l'analyse syntaxique

- ☐ `bin`contient le fichier binaire du compilateur nommé `tpcc`, qui est le résultat final de la compilation des fichiers sources. C'est le fichier exécutable qui peut être utilisé pour compiler les programmes écrits en tpc.

- ☐ `obj` contient les fichiers objets intermédiaires générés lors de la compilation des sources ainsi que le fichier `_anonymous.asm` qui correspond à la traduction du fichier programme tpc choisi.

- ☐ `rep` contient le rapport du script de test (`test.sh`)

- ☐ `test` est divisé en plusieurs sous répertoires contenant les jeux d'essais utilisés pour vérifier le bon fonctionnement du compilateur
  - ☐ `good` : programmes corrects sans avertissements
  - ☐ `syn-err` : erreurs lexicales / syntaxiques
  - ☐ `sem-err`: erreurs sémantiques
  - ☐ `warn` : programmes correctes avec avertissements

● Répartition du travail

Nous avons d’abord travaillé ensemble sur la conception et l'implémentation de la structure de données pour la table des symboles qui est essentielle pour avancer dans le projet.

Nous avons ensuite réparti le projet en deux parties : l’un de nous s’est concentré sur la vérification des erreurs sémantiques, tandis que l’autre s’est chargé de la traduction du code en nasm.

Chaque partie correspond à un parcours distinct de l’arbre syntaxique.

Le premier parcours effectue une vérification approfondie des erreurs sémantiques, assurant que le programme respecte les règles du langage tpc.

Le second parcours génère le code assembleur nasm, transformant le programme tpc en une version exécutable en assembleur.

Structure de Données de la Table de Symbole

- ## Définition des structures

  - ### Types de données

L'énumération `Type` définit les types de données que les symboles peuvent représenter, incluant les types de base (`INT`, `CHAR`, `VOID_`), les fonctions (`FUNCTION`), et un type par défaut (`DEFAULT`) utile pour les erreurs.
Le type `VOID` est utile pour la signature des fonctions.

  - ### Symbole

La structure `Symbol` représente un symbol dans le programme, contenant son nom (`ident`), son type (`type`), sa taille (`size`) qui dépend de son type, et son prochain emplacement en mémoire (`deplct`).

  - ### Table des Symboles

La structure `Symbols_Table` contient un tableau (de taille fixe = `INIT_SIZE`) de symboles (`tab`) et un index (`index`) indiquant la position du prochain symbole à ajouter. La taille d'un tableau est de 128, ce que l'on a jugé raisonnable.

  - ### Table des Fonctions

La structure `Function_Table` gère les informations sur les fonctions, incluant le nom (`ident`), le type de retour (`type_ret`), les tables des symboles pour les paramètres (`header`) et les variables locales (`body`), ainsi qu'un pointeur vers la prochaine fonction de la liste chainé (`next`).

- ### Table du programme

La structure `Program_Table` regroupe les tables des variables globales (`globals`) et les fonctions (`functions`)

• Fonctions associés

○ Initialisation

```
// INIT //

/* Create a symbol */
Symbol make_symbol(char* ident, Type type);

/* Initialize a symbol table */
Symbols_Table* init_Sym_table();

/* Initialize a function table */ // Empty header and body
Function_Table* init_Func_table();

/* Initialize a program table */
Program_Table* init_Program_table();
```

○ Libération de mémoire

```
64      // FREE //
65
66      /* Free a symbol */
67      void free_symbol(Symbol* symbol);
68
69      /* Free a symbol table */
70      void free_Sym_table(Symbols_Table* sym_table);
71
72      /* Free a function table */
73      void free_Func_table(Function_Table* func_table);
74
75      /* Free a program table */
76      void free_Program_table(Program_Table* prog_table);
```

- **Fonctions principales**

```
// CORE FUNCTIONS //

/* Return the type of the expr followed by the affection node
 * DEFAULT if assembling wrong types
*/
Type expr_type(Program_Table* program, Function_Table* table, Node * node, int Lvalue, int boolean);

/* Add a symbol to the symbol table
 * we suppose that the table as enough space left for the symbol
 * Return 1 if a problem occured, 0 otherwise
*/
int add_symbol(Symbols_Table* sym_table, Symbol symbol);

/* Add symbols to a symbol table
 * Returns 1 if a problem occured, 0 otherwise
*/
int add_Symbols_to_Table(Node *node, Symbols_Table * table);

/* Add function to a function table
 * Returns 1 if a problem occured, 0 otherwise
*/
int add_Function(Node *node, Function_Table * table, Program_Table * program);

/* Create the symbol table of a tree
 * Returns 1 if a problem occured, 0 otherwise
*/
int treeToSymbol(Node *node, Program_Table * table);
```

Ces structures et fonctions permettent de gérer efficacement les symboles et les fonctions facilitant ainsi la détection des erreurs sémantiques.

Gestion des Erreurs Sémantiques

- ## Détection des Erreurs

    - ### Variable non Déclarées

Lorsqu'un identifiant (`ident`) est rencontré dans l'arbre syntaxique après la déclaration de variable, le programme vérifie s'il est déjà défini dans l'un des tableaux de symboles. Si l'identifiant n'est pas trouvé, une erreur sémantique est levée pour indiquer sur l'identifiant n'est pas défini.

    - ### Vérification du Type des Expressions

La fonction `expr_type` est utilisée pour déterminer et vérifier le type des expressions dans le programme. Elle effectue des vérifications de type pour différents types d'expressions, telles que les opérations arithmétiques, les appels de fonctions et les expressions logiques.

    - ### Vérification des Types de Retour des Fonctions

La fonction `get_return_type` analyse les nœuds de retour (`return`) dans le corps d'une fonction pour s'assurer que les types de retour correspondent au type de retour déclaré de la fonction.

    - ### Vérification des Arguments des Fonctions

Les fonctions `count_args` et `function_parameters` vérifient que les arguments passées aux fonctions correspondent en type et en nombre aux paramètres attendus par les fonctions.

- # Difficultés rencontrées

    - ## Tableaux

Lors de la gestion des tableaux dans l'arbre syntaxique, nous faisons face à un problème particulier : différencier un entier d'un tableau. Un tableau est représenté par la présence d'un nœud fils indiquant sa taille. Cependant, lorsque l'on utilise un tableau sans préciser sa taille, comme dans la signature d'une fonction ou en tant que paramètre de fonction, la distinction entre un entier et un tableau devient ambiguë.

int fonction(int tab[], int number);

Dans cet exemple, les tableaux et les entiers sont représentés de la même manière dans l'arbre syntaxique. Pour résoudre ce problème, nous comparons la taille des symboles des variables : si la taille du symbole est supérieure à 4, il s'agit d'un tableau, sinon c'est un entier. Cependant, cette méthode n'est efficace que pour les tableaux de taille supérieure à 1.

    - ## Erreurs sémantiques ?

Une autre difficulté majeure consiste à différencier les erreurs sémantiques, les avertissements (warning) et les erreurs de programmation. Chacune de ces catégories d'erreurs doit être manière appropriée pour assurer la robustesse du compilateur et fournir des messages d'erreurs utiles aux développeurs.

Génération du code en asm

La traduction du code en assembleur n'a pas été terminée par manque de temps, la traduction des fonctions, et des variables locales n'ont pas été implémentés.

Nous n'avons pas rencontré de difficulté particulière, les TP permettant d'avancer pas à pas dans la traduction. Cependant comme nous ne passons pas à 100% le banc tests automatique de sémantique, nous n'avons pas pu essayer le banc de test automatique d'exécution, et nous n'avons pas pris le temps de tester le code traduit. Mais nous nous assurions auprès de nos chargées de TD que notre manière de traduire était cohérente.

Les traductions des fonctions `getchar()`, `getint()`, `putint()`, `putchar()` sont ajoutées en début de programme asm. C'était un choix temporaire, mais que nous n'avons pas eu le temps de corriger. L'idéal aurait été d'ajouter le code de ces fonctions uniquement dans le cas où elles sont effectivement appelées.

En début de traduction nous avions une fonction pour chaque instruction ou expressions différentes à traduire. Certaines parties du code se répétaient beaucoup alors nous avons fusionné certaines fonctions ressemblantes, comme les traductions d'opérateurs ou de comparaison et d'égalité. Cela complexifie légèrement le code C mais évite beaucoup de redondance.

Dans cette même idée nous avons fait le choix de rajouter des espacements et des commentaires pour presque chaque fonction de traduction. Cela complexifie légèrement le code c, mais simplifie énormément la lecture du code produit en NASM.

Interface Utilisateur

• Ligne de commande

Notre compilateur supporte plusieurs options de ligne de commande. Voici les options disponibles :

- `-t`, `--tree` affiche l'arbre syntaxique après l'analyse syntaxique
- `-h`, `--help` affiche l'aide et la description des options
- `-s`, `--symtabs` affiche la table des symboles après l'analyse sémantique

Le programme s'exécute à partir de la ligne de commande en utilisant le format suivant :

./bin/tpcc < fichier_entree.tpc [options]

Le fichier d'entrée doit être un fichier source tpc valide. Le compilateur génère les sorties en fonction des options spécifiées.

• Valeurs de Retour

Le programme utilise les codes de retour de suivants pour indiquer le résultat de l'exécution :

- ☐ `0` : Succès - L'analyse syntaxique et la compilation se sont déroulées avec succès
- ☐ `1` : Erreur de syntaxe - Le programme source contient une / des erreur(s) syntaxique(s)
- ☐ `2` : Erreur sémantique - Le programme source contient une / des erreur(s) sémantique(s)

Script de Test

Un script de test est fourni pour automatiser l'exécution du compilateur sur un ensemble de fichiers de test et vérifier les erreurs sémantiques :

./test.sh

Autres difficultées rencontrées

Lors de l'ajout d'une batterie de tests plus étendue, nous avons rencontré des erreurs syntaxiques inattendues. Ces tests, censés fonctionner correctement, ne passent pas, et la cause de ces échecs reste inconnue.
Cependant, en copiant le contenu du test dans un nouveau fichier, le test fonctionne correctement. Etant donné la quantité de tests, il ne nous restait plus le temps de tout recopier manuellement.

Conclusion

Ce projet de compilation couvre l'analyse lexicale, syntaxique et sémantique ainsi que la génération de l'arbre syntaxique et des tables des symboles. Malgré les défis rencontrés, comme la distinction entre les entiers et les tableaux ou notre série de tests qui ne fonctionnent pas, notre outil fournit des diagnostics clairs et une interface en ligne de commande efficace. Le script de test automatique facilite le débogage et renforce la fiabilité du compilateur.

Annexe

- **Définition des structures**

```c
typedef enum Type
{
    INT,
    CHAR,
    VOID_,
    FUNCTION,
    DEFAULT
} Type;
```

```c
typedef struct Symbol
{
    char* ident;        /* name of the symbol */
    Type type;          /* type of the symbol */
    int size;           /* size (in bytes) based on the type */
    long int deplct;    /* next space in memory */
} Symbol;
```

```c
typedef struct Symbols_Table
{
    Symbol tab[INIT_SIZE];      /* array of symbols */
    long int index;             /* index of the next symbol */
} Symbols_Table;
```

```c
typedef struct Function_Table
{
    char* ident;                    /* name of the function */
    Type type_ret;                  /* type */
    Symbols_Table* header;          /* parameters */
    Symbols_Table* body;            /* local variables */
    struct Function_Table* next;    /* next function */
} Function_Table;
```

```c
typedef struct Program_Table
{
    Symbols_Table* globals;         /* global variables */
    Function_Table* functions;      /* functions */
} Program_Table;
```

- **Gestion des erreurs sémantiques**

```
case ident:
    err_line = node->lineno;
    if (!isPresent_all(table, node)) {
        if (strcmp(node->data.ident, "getint") == 0);
        else if (strcmp(node->data.ident, "getchar") == 0);
        else if (strcmp(node->data.ident, "putchar") == 0);
        else if (strcmp(node->data.ident, "putint") == 0);
        else    {fprintf(stderr, "Line %d -> Semantic Error : \"%s\" is not defined\n", err_line,
    }
    break;
```

```
Type expr_type(Program_Table* program, Function_Table* table, Node * node, int Lvalue, int boolean) {
    Type left, right;
    Symbol * symbol;
    err_line = node->lineno;
    switch (node->label) {

    case ident:
        return process_ident_expr_type(program, table, node, Lvalue, boolean);

    case num:
        printf("%d ", node->data.num);
        return INT;

    case addsubUnaire:
        printf("%c ", node->data.byte);
        return expr_type(program, table, FIRSTCHILD(node), 0, 0);

    case addsub:
        left = expr_type(program, table, FIRSTCHILD(node), 0, 0);
        printf("%c ", node->data.byte);
        right = expr_type(program, table, SECONDCHILD(node), 0, 0);
        if ((left == CHAR && right == INT) || (left == INT && right == CHAR))
            fprintf(stderr, "\nLine -> %d Warning : Operation between CHAR and INT variables\n", err_line);
        return INT;
```

(Ceci n'est pas la fonction complète)

```
Type get_return_type(Node *node, Function_Table * table, Program_Table * program, Type wanted, Type *last) {
    if (node->label == _return) {
        if (FIRSTCHILD(node))
            *last = expr_type(program, table, FIRSTCHILD(node), 0, 0);
        else
            *last = VOID_;
    }
    for (Node *child = FIRSTCHILD(node); child != NULL && table->type_ret != VOID_; child = child->nextSibling) {
        err_line = child->lineno;
        if (*last != DEFAULT) {
            if (*last == VOID_) {
                fprintf(stderr, "Line %d -> Semantic Error : Function \"%s\" returns a value of type \"void\"\n", er
                return DEFAULT;
            }
            if ((wanted == INT || wanted == CHAR) && (*last != INT && *last != CHAR)) {
                return DEFAULT;
            }
        }
        get_return_type(child, table, program, wanted, last);
    }
    return *last == VOID_ ? DEFAULT : *last;
}
```

```
/* Count the number of args used
 * Check the types of the arguments
 * `used_from` is the function from which the function is called
 * If the types are correct, return the number of args used
 * Else return -1
*/
int count_args(Node * node, Program_Table * program, Function_Table * function, Function_Table * used_from, int boolean);

/* Compare the number of args used (`count`)
 * with the number of args that the function takes (`signature`)
 * Return 1 if `signature` == `count`, else 0
*/
int function_parameters(Function_Table * table, int count);
```

(prototype des fonctions)

```
else if (function->header->tab[index - i].size == -8) { // array in function header
    if (child->label == num) {
        fprintf(stderr, "Line %d -> Semantic Error : Type of the argument \"%d\" in function \"%s\"\n", err_line, child->dat
        fprintf(stderr, "Line %d -> Expected : TAB, Actual : INT\n", err_line);
        return -1;
    }
    Symbol * symbol = find_Symbol(program->globals, child->data.ident);
    if (symbol == NULL) symbol = find_Symbol(used_from->body, child->data.ident);
    printf("debug 2: %s\n", child->data.ident);
    if (symbol->size <= 4) {
        fprintf(stderr, "Line %d -> Semantic Error : Identifier \"%s\" is not an array\n", err_line, child->data.ident);
        return -1;
    }
}
```