Rapport de Projet

Théorie des Langages et Compilation

Génération de code

Polytech Lyon

Réalisateurs :
Abir Hanned, Chafae Qallouj, Ashley Padayodi , Nouha Meqdad

Table des matières

0.1 Introduction

Ce projet constitue la **deuxième partie** du projet de Théorie des Langages et Compilation. Il consistait à compléter et implémenter plusieurs méthodes dans le fichier `CodeGenerator.java` fourni dans l'énoncé, afin de générer du code assembleur à partir d'un langage source simplifié (type TCL).

Objectifs du projet

L'objectif principal était de compiler correctement :

- **Expressions arithmétiques et logiques** : addition, multiplication, comparaisons, opérations booléennes, etc.
- **Instructions conditionnelles et boucles** : if, while, for, blocs, et affichage via `print`.
- **Gestion des tableaux dynamiques** : allocation, accès, modification de taille et affichage.
- **Appels de fonctions et récursivité** : définition de fonctions, passage d'arguments, retour de valeurs et gestion de la pile d'exécution.

Concepts abordés

Ce projet permet de mettre en pratique plusieurs concepts clés de la théorie des langages et de la compilation :

- Génération de code linéaire à partir de l'AST (Abstract Syntax Tree) du programme source.
- Gestion des registres et des scopes pour assurer l'intégrité des variables et des fonctions.
- Allocation dynamique et manipulation de tableaux en mémoire.
- Gestion des appels de fonctions, incluant la récursivité et la sauvegarde/restauration du contexte.

Organisation du rapport

Le rapport est structuré en deux parties :

1. **Méthodologie et architecture** : présentation de l'architecture du CodeGenerator, des registres, des scopes, de la gestion des expressions, des tableaux et des fonctions.
2. **Tests et résultats** : exemples de programmes TCL, génération de code assembleur et résultats obtenus à l'exécution.

Ce rapport détaille la conception, l'implémentation et les choix techniques réalisés, ainsi que les problèmes rencontrés et les solutions adoptées.

0.2 Partie 1 : Méthodologie

0.2.1 Architecture du CodeGenerator

La classe `CodeGenerator` parcourt l'arbre syntaxique abstrait (AST) pour générer le code assembleur correspondant. Elle gère la réservation des registres, le suivi des scopes, la traduction des expressions arithmétiques et logiques, des structures conditionnelles et boucles, ainsi que la gestion des tableaux et des appels de fonctions. Elle utilise également les types fournis par le premier groupe pour assurer la cohérence des opérations et des affectations. Tous ces composants collaborent pour produire un programme assembleur exécutable par le simulateur.

0.2.2 Gestion des registres et des scopes

Chaque valeur ou variable temporaire est associée à un registre via la méthode `newRegister()`. Cette approche permet d'éviter les conflits entre valeurs intermédiaires.

Les scopes sont gérés avec `enterScope()` et `exitScope()`. Chaque bloc ou fonction crée un scope local, ce qui permet :

— La déclaration de variables locales sans interférer avec d'autres scopes.

— Le contrôle de la portée des variables, empêchant leur accès hors du bloc.

Cette méthode assure une allocation sécurisée des registres tout en respectant strictement la portée des variables.

0.2.3 Expressions arithmétiques et logiques

Pour générer le code des expressions arithmétiques et logiques, nous avons suivi une approche uniforme et systématique :

— **Évaluation des sous-expressions :** chaque opérande d'une expression (gauche et droite) est évalué séparément. On visite récursivement l'AST pour obtenir le code correspondant à chaque sous-expression.

— **Réservation de registres :** chaque sous-expression retourne son résultat dans un registre spécifique. Après avoir évalué les deux opérandes, un nouveau registre est réservé pour stocker le résultat final de l'opération courante. Cette méthode garantit que les valeurs intermédiaires ne sont pas écrasées.

— **Application de l'opération :** selon le type d'opérateur (arithmétique, comparaison, égalité, logique ou négation), le code assembleur correspondant est généré. Par exemple, pour une négation logique, on effectue un XOR avec 1 pour inverser 0 et 1, tandis que pour les comparaisons, on utilise des sauts conditionnels vers des labels spécifiques afin de stocker 0 ou 1 dans le registre résultat.

Cette approche garantit que chaque expression est correctement évaluée et que son résultat est stocké de manière sûre et cohérente pour être utilisé dans les expressions plus complexes ou pour être retourné par une fonction.

0.2.4 Instructions conditionnelles et boucles

Pour gérer les instructions conditionnelles (`if`) et les boucles (`while`, `for`), le générateur de code suit une approche basée sur l'utilisation de labels et de sauts conditionnels :

— **Évaluation des conditions :** Chaque condition est évaluée et son résultat est stocké dans un registre. Ce registre est ensuite comparé à zéro pour décider du flux d'exécution.

— **Labels pour le contrôle du flux :** Des labels uniques sont créés pour marquer le début et la fin des blocs conditionnels ou des boucles. Dans le cas d'un `if-else`, deux labels sont utilisés : un pour le bloc `else` et un pour la fin du `if`.

— **Sauts conditionnels et inconditionnels :**
  — `CondJump` permet de sauter vers un label si la condition est fausse.
  — `JumpCall` permet de sauter inconditionnellement à la fin d'un bloc ou au début d'une boucle.

— **Gestion des scopes :** Chaque bloc (`block` ou boucle `for`) crée un nouveau scope pour gérer correctement les variables locales et éviter les conflits avec des variables de portée extérieure.

Ainsi, le générateur de code transforme chaque instruction conditionnelle ou boucle en un ensemble de labels et de sauts, simulant efficacement le contrôle de flux du langage source en assembleur.

0.2.5 Gestion des tableaux

Le générateur de code gère les tableaux de manière dynamique, en tenant compte à la fois de l'initialisation, de l'accès aux éléments et de l'affichage. Les principaux points sont les suivants :

— **Initialisation :** Lorsqu'un tableau est déclaré ou initialisé, de la mémoire est réservée dynamiquement. Un pointeur vers le tableau et sa taille sont stockés, ce qui permet d'accéder et de modifier ses éléments.

— **Accès et modification :** Pour lire ou écrire un élément `t[i]`, le générateur calcule l'adresse mémoire correspondante à partir du pointeur du tableau et de l'indice `i`. Si une affectation `t[i] = x` dépasse la taille actuelle du tableau, celui-ci est automatiquement redimensionné grâce à la méthode `resizeArrayIfNeeded()`, ce qui permet d'éviter les erreurs d'accès mémoire.
— **Registres dédiés :** Chaque opération sur un tableau utilise des registres pour stocker temporairement les adresses et les valeurs, garantissant que l'évaluation des expressions reste sûre et cohérente avec le reste du code généré.
— **Affichage :** La méthode `printArray` permet d'afficher les tableaux sous forme lisible, en parcourant tous les éléments et en générant les instructions assembleur correspondantes. Actuellement, cette méthode est limitée aux tableaux unidimensionnels (tableaux simples).

Ainsi, le générateur transforme les opérations sur tableaux du langage source en instructions assembleur sécurisées, tout en conservant la flexibilité nécessaire pour gérer des tailles dynamiques et des pointeurs.

0.2.6 Appels de fonctions et récursivité

Le générateur de code gère les appels de fonctions en respectant le contexte d'exécution de chaque fonction. Pour chaque appel, il procède comme suit :

— **Évaluation des arguments :** Les arguments de la fonction sont évalués de gauche à droite, chacun étant stocké dans un registre temporaire. Ces registres sont ensuite empilés sur la pile avant l'appel de la fonction.
— **Réservation du registre de retour :** Chaque fonction dispose d'un registre dédié pour stocker sa valeur de retour. Ce registre est préalloué avant l'exécution de la fonction pour éviter tout conflit avec d'autres fonctions ou appels récursifs.
— **Sauvegarde du contexte :** Avant l'appel d'une fonction récursive, tous les registres actifs (sauf le registre de retour) sont sauvegardés sur la pile afin de préserver l'état des appels précédents. Cette décision a été prise pour résoudre la difficulté de gérer la pile et les registres lors d'appels récursifs.
— **Instruction CALL/RET :** L'appel est généré via l'instruction `CALL`, et le retour via `RET`, garantissant que l'exécution reprend correctement après la fonction appelée.
— **Restauration du contexte :** Après le retour d'une fonction, les registres précédemment sauvegardés sont restaurés dans l'ordre inverse, ce qui permet de continuer l'exécution avec le même état qu'avant l'appel.

Cette organisation assure que chaque fonction peut être appelée de manière isolée, que les arguments et résultats sont correctement gérés, et que les appels

récursifs ne perturbent pas les registres ou variables locales.

0.3 Partie 2 : Tests et exemples

Les fichiers de test se trouvent dans le dossier `test/`. Chaque test contient deux fichiers principaux :

- **Code source TCL** : fichiers `TestX.txt` contenant le programme écrit en langage source simplifié.
- **Code assembleur généré** : fichiers `TestX.asm` correspondant au code assembleur produit par le générateur.

Cette organisation permet de lier facilement chaque programme source à son code assembleur intermédiaire, et de vérifier la correspondance entre le comportement attendu et l'exécution.

0.4 Tests et résultats

Dans cette section, nous présentons deux exemples illustrant la génération de code assembleur à partir du langage source TCL.

0.4.1 Exemple 1 : Factorielle récursive

Programme source TCL :

```
int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}

int main() {
    int result = fact(5);
    print(result);
    return 0;
}
```

Code assembleur généré (extrait simplifié) :

```
XOR R0 R0 R0
ADDi R1 R0 1
XOR R2 R2 R2
CALL main
```

```
STOP
main: ADDi R3 R0 5
ST R3 R2
ADDi R2 R2 1
ST R3 R2
ADDi R2 R2 1
CALL fact
LD R4 R2
SUBi R2 R2 1
SUBi R2 R2 1
LD R3 R2
PRINT R4
XOR R5 R5 R5
ADDi R5 R5 10
OUT R5
ADDi R6 R0 0
ST R6 R2
RET
fact: ADDi R0 R0 0
LD R8 R7
  ...
MUL R18 R13 R17
ST R18 R2
RET
```

Résultat à l'exécution :

```
120
```

0.4.2 Exemple 2 : Tableaux et boucles

Programme source TCL :

```
int main() {
    int[] numbers = {10, 25, 30, 15, 20};

    int sum = 0;
    int i = 0;
    while (i < 5) {
        sum = sum + numbers[i];
        i = i + 1;
```

```
    }
    print(sum);

    int max = 0;
    int j = 0;
    while (j < 5) {
        if (numbers[j] > max) {
            max = numbers[j];
        }
        j = j + 1;
    }
    print(max);

    numbers[0] = 100;
    numbers[4] = 200;
    print(numbers);

    return 0;
}
```

Code assembleur généré (extrait simplifié) :

```
XOR R0 R0 R0
ADDi R1 R0 1
XOR R2 R2 R2
CALL main
STOP
main:
  ADDi R5 R0 10
  ST R5 R1
  ADDi R6 R0 25
  ST R6 R1
  ...
StartWhile_1: ... EndWhile_2: PRINT R11
StartWhile_5: ... EndWhile_6: PRINT R28
ADDi R50 R0 100
ADDi R58 R0 200
PRINT array
RET
```

Résultat à l'exécution :

```
100
30
100 25 30 15 200
```

Ces deux exemples illustrent la capacité du générateur à gérer les fonctions récursives, les boucles, les conditions, ainsi que l'allocation et la manipulation de tableaux dynamiques.

0.4.3 Conclusion et perspectives

Ce projet a permis de générer correctement du code assembleur à partir d'un langage simplifié, en gérant expressions, boucles, tableaux et fonctions récursives. À l'avenir, il serait intéressant d'optimiser l'utilisation des registres et d'ajouter la gestion d'erreurs et de types plus complexes.