GROUPE 2
GÉNERATION DE CODE

Chafae QALLOUJ - Nouha MEQDAD - Abir HANNED - Ashley PADAYODI

3A INFORMATIQUE

LUNDI 5 JANVIER 2026

SOMMAIRE

1-INTRODUCTION

Génération de code :

- Parcours de l’arbre syntaxique abstrait (AST)
- Gestion des types et des variables via les informations du premier sous-groupe
- Scopes imbriqués avec pile de registres pour variables locales
- Registres illimités → allocation d’un registre par expression ou variable
- Génération de code assembleur pour toutes les structures du langage : expressions, boucles, conditions, fonctions, tableaux, appels, etc.
- Implémentation des méthodes visit pour chaque type de nœud de l’AST

2-GESTION DES VARIABLES

Stratégie : Pile de Scopes

Chaque scope :

✓ Enregistre ses variables locales
✓ Accède aux variables parentes
✓ Libère ses registres à la sortie

Avantages :

✓ Isolement des variables locales
✓ Pas de confusion avec variables du même nom
✓ Réutilisation efficace des registres

EXEMPLE :

```
int a = 5;    // Scope global :
              //   a → reg 0
if (true) {   // Scope local :
  int a = 10; //   a → reg 5
  print(a);   //   affiche 10
}

print(a);     // a → reg 0
              // affiche 5
```

3-EXPRESSIONS ARITHMÉTIQUES ET LOGIQUES

Expressions arithmétiques

- Constantes entières (visitInteger) : chargées directement dans un registre.
- Addition / Soustraction (visitAddition) : évaluation des deux opérandes → application de l'instruction ADD/SUB → résultat dans un nouveau registre.
- Multiplication / Division / Modulo (visitMultiplication) : même stratégie, instruction correspondante (MUL, DIV, MOD).
- Opposé arithmétique (visitOpposite) : calcul de 0 moins la valeur de l'expression.

3-EXPRESSIONS ARITHMÉTIQUES ET LOGIQUES

Expressions logiques et booléennes

- Booléens (visitBoolean) : false → 0, true → 1
- Négation (visitNegation) : XOR avec 1 → 0 devient 1, 1 devient 0
- AND / OR (visitAnd / visitOr) : évaluation des deux opérandes → application directe des instructions AND / OR sans sauts conditionnels

4-BOUCLES ET CONDITIONS

✓ **LES COMPARAISONS(› ‹ ›= ‹= == !=) :** visitComparison, visitEquality

Registre résultat (1 =vrai, 0= faux)

✓ **if, *while et for* :** utilisent directement ce résultat.  Si la condition est fausse, on saute vers un label de fin, sinon on exécute le bloc.

✓ **LES BLOCS{} :**

- Chaque bloc introduit un nouveau scope(gérer la durée de vie des variables).
- Les variables sont locales au bloc.
- Les registres associés aux ces variables ne sont plus utilisés à la sortie du bloc.

5-GESTION DES TABLEAUX

Initialisation

public Program visitTab_initialization (ctx)

- Allocation d’un bloc mémoire de 12 cases
  - Longueur du tableau
  - 10 éléments du tableau
  - Pointeur vers le prochain bloc
- Stockage de la longueur
- Ajout des éléments avec une boucle
- En cas de dépassement de bloc, allocation d’un nouveau bloc

private Program allocateBlock()

- Heap Pointer pour connaître l’adresse de la prochaine case libre en mémoire
- Allocation d’un bloc mémoire de 12 cases
- Toutes les cases sont initialisées à 0

5-GESTION DES TABLEAUX

Accès et modification

public Program visitTab_access(ctx)

- Erreur en cas de tentative d’accès à un emplacement mémoire non alloué au tableau (SegFault)

public Program visitAssignment(ctx)

- Traitement du cas d’un tableau avec la méthode resizeArrayIfNeeded(tabReg, indexReg)

private Program resizeArrayIfNeeded(tabReg, indexReg)

- Alloue de nouveaux blocs si besoin
- Initialise les éléments du tableau à 0
- Met à jour la longueur du tableau

5-GESTION DES TABLEAUX

Affichage

public Program visitPrint(ctx)

- Traitement du cas d’un tableau avec la méthode printArray(tabReg)

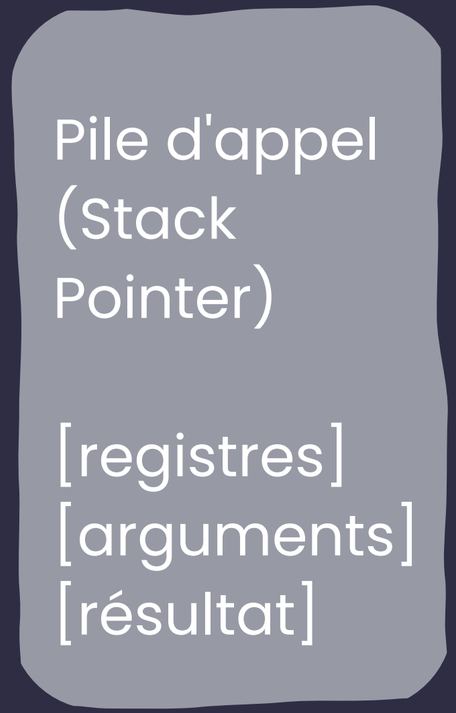
private Program printArray(tabReg)

- Affichage simplifié
  - Exemple : 1 2 3 4 5
- Boucle sur les éléments en prenant en compte le fait que le tableau soit stocké dans des blocs de 10 cases (possiblement disjoints)

6-GESTION DES FONCTIONS

Stratégie : Sauvegarde/restauration

Déclaration de fonction :
✓ Label unique pour l'adresse de la fonction
✓ Création d'un nouveau scope
✓ Chargement des paramètres depuis la pile
✓ Exécution du corps
✓Instruction return avec résultat sur pile



Appel de fonction :
✓ Évaluation des arguments
✓Sauvegarde des registres actifs
✓ Empilage des arguments
✓CALL vers le label de la fonction
✓ Récupération du résultat
✓ Restauration des registres (ordre inverse)

6-GESTION DES FONCTIONS

Corps de fonction (core_fct) :

✓ Évaluer chaque instruction du corps

✓ Si instruction return explicite :

→ visitReturn() traite l'expression

✓ Si pas de return explicite :

→ Évaluer la dernière expression (si elle existe)

→ Stocker la valeur sur la pile

→ Exécuter RET (retour implicite)

Return explicite :

✓ Évaluer l'expression de retour

✓ Stocker le résultat sur la pile (SP)

✓ Exécuter RET immédiatement

7-CONCLUSION

Le code fonctionne pour les parties essentielles : expressions, comparaisons, conditions, boucles, fonctions et tableaux simples.

La gestion des scopes est correcte et évite les erreurs sur les variables.

En revanche, certaines fonctionnalités avancées ne sont pas gérées, comme les tableaux multidimensionnels ou les optimisations.

MERCI POUR VOTRE ÉCOUTE.

Chafae QALLOUJ - Nouha MEQDAD - Abir HANNED - Ashley PADAYODI

3A INFORMATIQUE