LOG3210 - Éléments de langages et compilateurs
TP3 : Génération de code intermédiaire

Hubert Boucher
Thierry Beaulieu

Hiver 2025

1 Objectifs

- Se familiariser avec la traduction dirigée par la syntaxe (SDT);
- Utiliser une forme de code intermédiaire (le code à trois adresses);
- Implémenter des techniques d'optimisation de code intermédiaire (Fall-Through).

2 Contexte

Jusqu'ici, nous avons réalisé l'analyse lexicale et syntaxique (TP1) et l'analyse sémantique (TP2). Pour rappel:

- **L'analyse lexicale** a permis de confirmer que tous les mots de notre code sont bien écrits;
- **L'analyse syntaxique** a vérifié que le texte et les phrases suivent une bonne structure;
- **L'analyse sémantique** a vérifié que les phrases avaient du sens.

Une fois ceci complété, on peut avoir confiance en le fait que le code est valide.

3 Travail à faire

Maintenant, la prochaine étape du compilateur est de faire quelque chose avec ce code. Le TP3 consiste donc à écrire un visiteur qui traduira du code valide en une représentation intermédiaire, et qui fera certaines optimisations simples sur celui-ci. **Par exemple** (pour `test_condwhile_1.txt`)

Code (validé grâce aux TP1 et TP2)

```
while (b) {
   a = 42;
}
```

(où `b` est une variable booléenne)

⇒

Code intermédiaire (résultat du TP3)

```
_L1
if b == 1 goto _L2
goto _L0
_L2
a = 42
goto _L1
_L0
```

La représentation intermédiaire choisie est le code à trois adresses, similaire à celle présentée dans le livre du dragon au chapitre 6. La différence majeure avec le livre est que les labels seront intercalés avec le code plutôt que mis à part. De plus, seules les parties concernant les expressions arithmétiques et les flux de contrôle seront utilisées.

Le livre utilise dans ses exemples une approche où il concatène la chaîne de caractères représentant le code intermédiaire. Dans ce TP, il sera plus simple de "print" le résultat au fur et à mesure.

Traduction des règles du livre en Java:

- Les `gen(...)` ou `label(...)` deviennent `m_writer.println(...)`
- `code`: visite/accepte le noeud (`jjtAccept`)

Note: Dans ce TP3, on peut considérer que l'analyseur sémantique a déjà bien fait le travail de vérifier les éventuelles erreurs. Il n'est donc pas nécessaire de contrecarrer des erreurs rendu à cette étape de la compilation. Techniquement, il serait possible de réaliser le TP2 et TP3 dans le même visiteur, mais cela alourdirait le visiteur inutilement.

Vous devrez, pour votre part, implémenter deux visiteurs, `IntermediateCodeGenVisitor` et ensuite `IntermediateCodeGenFallVisitor`, répondant aux prochaines sections.

3.1 Code intermédiaire sans fall-through

Dans un premier temps, il faut traduire les expressions de la grammaire en code à trois adresses tel que décrit dans le document de référence ***SDD pour la génération de code intermédiaire (v3.b)*** (aussi repris en annexe de ce TP).

Cette première implémentation est à faire dans le visiteur `IntermediateCodeGenVisitor`. Nous vous conseillons de faire passer les tests dans l'ordre suivant:

1. `test_arith_*.txt` (expressions arithmétiques)
2. `test_bool_*.txt` (expressions booléennes)
3. `test_compare_*.txt` (comparaisons)
4. `test_condif_*.txt` (if-else)
5. `test_condwhile_*.txt` (boucles while)
6. `test_for_*.txt` (boucles for)
7. `test_script_*.txt` (scripts)

Note pour les `for`

Les `for` ne sont pas inclus dans la SDD! Vous pouvez vous servir de la sortie des tests pour comprendre la logique de la boucle `for`, et imaginer son implémentation dans le visiteur. Aidez-vous aussi du fichier `Langage.jjt` pour comprendre comment la boucle devrait être visitée.

Note pour les `scripts`

Jusqu'à maintenant, le code qu'on a analysé était une simple suite de blocs. Ici, nous allons aussi permettre aux utilisateurs de notre langage de nommer certaines parties de notre code (que l'on appellera des scripts). Ces scripts pourront ensuite être appelés à la fin du fichier. Exemple:

```
script1 {
   a = 1;
}

script2 {
   c = 3;
}

script3 {
   b = 2;
}

script3()
script1()
```

Nous définirons donc toujours des scripts en premier lieu (`ASTScript`). Ensuite, à la fin du code, ces scripts sont appelés (`ASTScriptCall`). Votre but sera de visiter ces nouveaux nœuds, afin de générer le code intermédiaire **dans l'ordre des appels des blocs**.
Les scripts qui ne sont pas appelés ne peuvent pas se retrouver dans le code généré! Et des scripts appelés plusieurs fois doivent évidemment se retrouver autant de fois qu'appelés.
Conseil: Référez-vous au `Langage.jjt` pour comprendre la logique de lecture des scripts! Notamment pour les `ASTScriptCall`.
Attention! Nous n'accepterons pas l'usage d'une variable globale à la classe pour stocker l'ordre d'exécution.

3.2 Code intermédiaire avec fall-through

Dans un deuxième temps, vous devrez étendre votre code pour couvrir le fall-through. Les fall-through sont une optimisation du code intermédiaire de manière à éliminer certains `goto` redondants.

Remarque

Il n'est pas attendu que le code généré soit parfaitement optimisé (élimination de code mort...). De telles optimisations demandent des optimisations successives qui sortent du cadre de ce cours.

Cette deuxième implémentation est à faire dans le visiteur `IntermediateCodeGenFallVisitor`. Vous pouvez copier/coller votre code de `IntermediateCodeGenVisitor` et repartir de là, ou utiliser l'héritage Java. Travaillez bien dans le deuxième fichier, en gardant le premier intact!

4 Conseils importants

Voici une liste de conseils pour vous aider pour ce TP:

- Faites bien le TP dans l'ordre de la section 3.1!

- Faites bien le TP dans l'ordre de la section 3.1!
- Dans le code source, des objets/classes vous sont fournies pour vous aider:
  - `newID()`: génère une variable temporaire (type `_t*`);
  - `newLabel()`: génère un label (type `_L*`);
  - `class BoolLabel {...}`: représente un label booléen pour un noeud de l'arbre (composé d'un label `true` et d'un label `false`, voir SDD).
- Utilisez `data` pour communiquer **du parent vers l'enfant**. En général ce seront des objets `BoolLabel`.
- Utilisez les `return` pour communiquer **de l'enfant vers son parent**. En général, ce sont des `String` représentant une adresse (nom de variable, valeur entière, variable temporaire).
- Le correctif a été fait en suivant la table SDD donnée en annexe, suivez la bien!
- Ne faites pas d'optimisations complémentaires.

5 Barème

Le TP est évalué sur 20 points, distribués comme suit :

Fonctionnalités	
Code intermédiaire pour les expressions arithmétiques	/3
Code intermédiaire pour les expressions booléennes	/3
Code intermédiaire pour les if-else et boucles while	/3
Code intermédiaire pour les boucles for	/3
Code intermédiaire pour les scripts	/4
Code intermédiaire avec Fall-Through	/4
Pénalités	
Retard (par jour de retard)	-10
Non-respect des consignes de remise (nom du fichier,...)	-4
Code de mauvaise qualité	-2
Total	**/20**

L'ensemble des tests doivent passer au vert pour que le laboratoire soit réussi.

6 Remise

L'échéance pour la remise du TP3 est le `30 Mars 2025 à 23:59`.

Remettez sur Moodle une archive nommée `log3210-tp3-matricule1-matricule2.zip` (tout en minuscule), contenant **uniquement** les 3 fichiers suivants:

- `IntermediateCodeGenVisitor.java`;
- `IntermediateCodeGenFallVisitor.java`;
- `README.md` (facultatif, à fournir si vous avez des commentaires à ajouter sur votre projet).

Le devoir doit être fait en **binôme**. **Les remises individuelles ne sont pas autorisées !** Si vous ne trouvez pas de binôme, veuillez nous contacter au plus vite (minimum 5 jours avant la remise).

Si vous avez des questions, veuillez nous contacter sur Discord via le canal de votre équipe (`#equipe-00`), sans nous mentionner! Les chargés vous répondront dès que possible, s'ils ont du temps libre en dehors des séances!

A Aide: SDD

N.B.: E correspond à une expression numérique et B une expression booléenne. Donc si deux règles de production semblent pareil, il faut vérifier le type de l'expression.

A.1 Initial

Code intermédiaire pour les expressions arithmétiques

Production	Semantic rules
$E \to E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \| E_2.code \| gen(E.addr' =' E_1.addr' +' E_2.addr)$
$E \to -E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \| gen(E.addr' ='' \textbf{minus}' E_1.addr)$
$E \to (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$E \to \textbf{id}$	$E.addr = top.get(id.lexeme)$ $E.code =''$

Code intermédiaire pour les expressions booléennes

Production	Semantic rules
$B \to B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.true) \| B_2.code$
$B \to B_1 \| B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.false) \| B_2.code$
$B \to !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \to E_1\ \textbf{rel}\ E_2$	$B.code = E1.code \| E2.code$ $\quad \| gen('if' E_1.addr\ \textbf{rel}.op\ E_2.addr 'goto' B.true)$ $\quad \| gen('goto' B.false)$
$B \to true$	$B.code = gen('goto' B.true)$
$B \to false$	$B.code = gen('goto' B.false)$

$B \rightarrow \textbf{id}$	$B.code = gen('if' top.get(id.lexeme)' == 1 goto' B.true) \| gen('goto' B.false)$

Code intermédiaire pour les statements

Production	Semantic rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \| label(S.next)$
$S \rightarrow id = B$	$B.true = newlabel()$ $B.false = newlabel()$ $S.code = B.code \| label(B.true) \| gen(top.get(id.lexeme)' = 1') \| gen('goto' S.next) \|$ $label(B.false) \| gen(top.get(id.lexeme)' = 0')$
$S \rightarrow id = E$	$S.code = E.code \| gen(top.get(id.lexeme)' =' E.addr)$
$S \rightarrow \textbf{if}(B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$
$S \rightarrow \textbf{if}(B) S_1 \textbf{else} S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$ $\quad \| gen('goto' S.next) \| label(B.false) \| S_2.code$
$S \rightarrow \textbf{while}(B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| B.code \| label(B.true)$ $\| S_1.code \| gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \| label(S1.next) \| S_2.code$

A.2 Fall through

Code intermédiaire pour les expressions booléennes

Production	Semantic rules
$B \rightarrow B_1 \&\& B_2$	$B_1.true =$ "$fall$" $if(B.false ==$ "$fall$") $\quad B_1.false = newlabel()$ $else$ $\quad B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $if(B.false ==$ "$fall$") $\quad B.code = B_1.code \| B_2.code \| label(B_1.false)$ $else$ $\quad B.code = B_1.code \| B_2.code$
$B \rightarrow B_1 \| B_2$	$if(B.true ==$ "$fall$") $\quad B_1.true = newlabel()$ $else$ $\quad B_1.true = B.true$ $B_1.false =$ "$fall$" $B_2.true = B.true$ $B_2.false = B.false$ $if(B.true ==$ "$fall$") $\quad B.code = B_1.code \| B_2.code \| label(B_1.true)$ $else$ $\quad B.code = B_1.code \| B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$if(B.true! =$ "$fall$"$\&\& B.false! =$ "$fall$") $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('if' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.true)$ $\quad\quad \| gen('goto' B.false)$ $elif(B.true! =$ "$fall$"$\&\& B.false ==$ "$fall$") $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('if' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.true)$ $elif(B.true ==$ "$fall$"$\&\& B.false! =$ "$fall$") $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('ifFalse' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.false)$ $else$ $\quad$ error
$B \rightarrow true$	$if(B.true! =$ "$fall$") $\quad B.code = gen('goto' B.true)$ $else$ $\quad B.code = ''$

$B \rightarrow false$	$if(B.false! = \text{“}fall\text{”})$ $\quad B.code = gen('goto' B.false)$ $else$ $\quad B.code = ''$
$B \rightarrow \textbf{id}$	$if(B.true! = \text{“}fall\text{”} \&\& B.false! = \text{“}fall\text{”})$ $\quad B.code = gen('if' top.get(id.lexeme)' == 1 goto' B.true) \| gen('goto' B.false)$ $elif(B.true! = \text{“}fall\text{”} \&\& B.false == \text{“}fall\text{”})$ $\quad B.code = gen('if' top.get(id.lexeme)' == 1 goto' B.true)$ $elif(B.true == \text{“}fall\text{”} \&\& B.false! = \text{“}fall\text{”})$ $\quad B.code = gen('ifFalse' top.get(id.lexeme)' == 1 goto' B.false)$ $else$ $\quad$ error

Code intermédiaire pour les statements

Production	Semantic rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \| label(S.next)$
$S \rightarrow id = B$	$B.true = \text{“}fall\text{”}$ $B.false = newlabel()$ $S.code = B.code \| gen(top.get(id.lexeme)' = 1') \| gen('goto' S.next) \|$ $label(B.false) \| gen(top.get(id.lexeme)' = 0')$
$S \rightarrow \textbf{if}(B) S_1$	$B.true = \text{“}fall\text{”}$ $B.false = S_1.next = S.next$ $S.code = B.code \| S_1.code$
$S \rightarrow \textbf{if}(B) S_1 \textbf{else} S_2$	$B.true = \text{“}fall\text{”}$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \| S_1.code$ $\quad \| gen('goto' S.next) \| label(B.false) \| S_2.code$
$S \rightarrow \textbf{while}(B) S_1$	$begin = newlabel()$ $B.true = \text{“}fall\text{”}$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| B.code$ $\| S_1.code \| gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \| label(S1.next) \| S_2.code$