LOG3210 - Élément de langages et compilateur
TP3 : Génération de code intermédiaire

Doriane Olewicki – Chargée de cours
Esther Guerrier – Chargée de laboratoire

Hiver 2021

1 Objectifs

- Se familiariser avec la traduction dirigée par la syntaxe
- Utilisé une forme de code intermédiaire : le code à trois adresses;
- Implémenter des techniques d'optimisations de code intermédiaire.

2 Travail à faire

Une fois l'analyse lexical, syntaxique et sémantique complétée, on peut avoir confiance en le fait que le code validé par ses analyses est valide. L'analyse lexical à 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, et l'analyse sémantique a vérifié que les phrases avaient un sens.

La prochaine étape du compilateur est de faire quelque chose avec ce code. Le troisième travail pratique consistera à écrire un visiteur qui traduira du code valide en une représentation intermédiaire, et qui fera certaine optimisation simple sur celui-ci.

La représentation intermédiaire choisis 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, seule la partie concernant les expressions arithmétique et les flux de contrôles sera utilisée.

Le livre utilise dans ses exemples une approche ou il concatène la chaine de charactère 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:

- "gen" ou "label": deviennent "print""
- "code": visiter/accepté le noeud ("jjtAccept")

Dans tous le TP, on peut considérer que les tests d'analyse sémantique passent déjà. Il n'est pas nécessaire de contrecarrer des erreurs rendu à cette étape de la compilation, grâce aux vérifications des étapes précédentes. Il serait possible de faire l'analyse sémantique et la génération de code intermédiaire en une seul passe, mais cela alourdirait le visiteur inutilement: pour le bien de ce Travail Pratique, on ne fera pas les deux. Une implémentation du visiteur `SemantiqueVisitor` est fournis avec les sources. Il est donc conseillé de tester son générateur de code intermédiaire en fournissant les mêmes données aux deux visiteurs en parallèle, de sorte que si le code tester est invalide, l'analyseur syntaxique le révèlera aussitôt.

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

2.1 Code intermédiaire initial

Il faut traduire les expressions de la grammaire en code à trois adresses tel que décris dans le document de référence ***SDD pour la generation de code intermediaire (v3.b)***.

Vous devez traduire les expressions :

- Arithmétique (expression sur des nombres avec des opérantes);
- Booléennes (expressions sur des nombres ou booléen avec des opérantes);
- Conditionnelles (if et while).

Dans le document de référence, vous trouverez le SDD à appliquer pour ce TP.

2.1.1 Switch

Vous devez aussi implémenté la **traduction des switchs**. Pour cet exercice, vous devez implémenter l'écriture du code intermédiaire pour l'expression switch, sans avoir les règles sémantiques fournies. Pour cela, suivez les consignes des slides du cours 7.

Voici un exemple de la représentation machine :

Listing 1: Switch statement

```
switch(NUM0) {
case NUM1 : stmt1
case NUM2 : stmt2
default : stmt3
}
```

Listing 2: Code intermédiaire

```
goto _L1
_L2
//stmt1 traduction
goto _L0
_L3
//stmt2 traduction
goto _L0
_L1
if NUM0 == NUM1 goto _L2
if NUM0 == NUM2 goto _L3
_L0
```

2.2 Code intermédiaire avec fall-through

Dans un deuxième temps, vous devrez étendre votre code pour couvrir le fall-through (optimisation du code intermédiaire pour les booléens). Le code est aussi dans le document de référence.

Cette réduction est aussi appelé élimination des GOTO redondants.

Il est possible de réduire le nombre de label et de goto en utilisant la technique présentée à la section 6.6.5 du livre du dragon. Cette technique doit bien entendu être adapté pour notre grammaire. La référence de variable fonctionne de la même manière que présenté à la figure 6.39, pour l'utilisation d'opérateur relationnelle. Le livre du dragon vous fournira pas mal d'explication sur les différentes choses à implémenter, mais **je vous sugère de vous référez principalement au document "SDD pour la generation du code intermediaire (v3)"** sur moodle, section 3, "style fall through".

Cette technique n'éliminera pas une partie des goto et labels qu'il serait possible d'éliminer à la main. Il n'est pas attendu que le code généré soit totalement optimisé.Il est normal qu'il y ait du code mort, des labels inutiles et des "goto" redondant, même après cette optimisation. L'optimisation parfaite du code représenterais une tache beaucoup plus grande que celle demandé dans le TP et demanderais plusieurs techniques successives d'optimisation qui ne sont pas à l'étude pour ce cours d'introduction. Je vous demande de ne pas implémenter plus d'optimization que celle du fall through.

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

3 Conseils importants

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

- Commencé par la partie arithmétique, ensuite les expressions booléennes, ensuite les conditions.
- Dans le code source, des objets/classes vous sont fournies pour vous aider.
  - `genId()`: génère une variable temporaire (type "_t*");
  - `genLabel()`: génère une label (type "_L*");
  - `Class BoolLabel`: représente les label booléen pour un noeud de l'arbre (avec le label "true", et le label "false", voir SDD).
- Utilisez `data` pour communiquer du parent vers l'enfant, en général ce seront des object `BoolLabel`.
- Utiliser la valeur de return pour communiquer de l'enfant vers son parent, en général une `String` représentant une adresse (nom de variable, valeur entière, variable temporaire).
- Le correctif à été fait en suivant le SDD donné en annexe, suivez le bien !!
- Ne faites pas d'optimisation complémentaire.
- Quand vous avez fini le code sans Fall-Through, copié l'entièreté du code dans la version avec Fall-Through et repartez de là!

4 Barème

Le TP est évalué sur 20 points, les points étant distribué comme suit :

- Code intermédiaire pour les expressions arithmétiques : 2 points
- Code intermédiaire pour les expressions booléennes : 3 points
- Code intermédiaire pour les statements conditionnelles : 3 points
- Code intermédiaire pour les switchs : 3 points
- Code intermédiaire avec Fall-Through : 6 points
- Test surprise : 2 points
- Qualité du code : 1 point

L'ensemble des tests donnés sous Ant doivent passer au vert pour que le laboratoire soit réussi. La qualité du code sera aussi vérifiée et prise en compte pour la correction.

5 Remise

Le devoir doit être fait en **binôme**. Remettez sur Moodle une archive nommée *log3210-tp3-matricule1-matricule2.zip* avec uniquement les fichiers `IntermediateCodeGenVisitor.java` et `IntermediateCodeGenFallVisitor.java` ainsi qu'un fichier `README.md` contenant tous commentaires concernant le projet. **Note: remplir ce fichier n'est pas nécessaire. Remplissez-le seulement si vous avez des commentaires à rajouter sur votre code.**

L'échéance pour la remise est le **21 mars 2021 à 23 h 55**.

Une pénalité de 10 points (50%) s'appliquera par jour de retard. Une pénalité de 4 points (20%) s'appliquera si la remise n'est pas conforme aux exigences (nom du fichier de remise).

Les remises individuelles ne sont pas autorisées ! Si vous ne trouvé par de binôme, veuillez m'envoyer un email au plus vite (minimum 5 jours avant la remises).

Si vous avez des questions, veuillez me contacter sur discord ou, pour une urgence, sur mon courriel : esther.guerrier@polymtl.ca.

5.1 Aide: SDD

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

5.1.1 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
$B \to B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.true) \| B_2.code$
$B \to B_1 \| B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \| label(B_1.false) \| B_2.code$
$B \to !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \to E_1$ **rel** E_2	$B.code = E1.code \| E2.code$ $\| gen('if' E_1.addr$ **rel**$.op\ E_2.addr' goto' B.true)$ $\| gen('goto' B.false)$
$B \to true$	$B.code = gen('goto' B.true)$
$B \to false$	$B.code = gen('goto' B.false)$

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

Code intermédiaire pour les statements

Production	Semantic rules
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \| label(S.next)$
$S \rightarrow id = B$	$B.true = newlabel()$ $B.false = newlabel()$ $S.code = B.code \| label(B.true) \| gen(top.get(id.lexeme)' = 1') \| gen('goto'S.next) \|$ $label(B.false) \| gen(top.get(id.lexeme)' = 0')$
$S \rightarrow id = E$	$S.code = E.code \| gen(top.get(id.lexeme)' =' E.addr)$
$S \rightarrow \textbf{if}(B)S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$
$S \rightarrow \textbf{if}(B)S_1\textbf{else}S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code \| label(B.true) \| S_1.code$ $\quad \| gen('goto'S.next) \| label(B.false) \| S_2.code$
$S \rightarrow \textbf{while}(B)S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| B.code \| label(B.true)$ $\| S_1.code \| gen('goto'begin)$
$S \rightarrow S_1S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \| label(S1.next) \| S_2.code$

5.1.2 Fall through

Code intermédiaire pour les expressions booléennes

Production	Semantic rules
$B \rightarrow B_1 \&\& B_2$	$B_1.true =$ "$fall$" $if(B.false ==$ "$fall$") $\quad B_1.false = newlabel()$ $else$ $\quad B_1.false = B.false$

	$B_2.true = B.true$ $B_2.false = B.false$ $if(B.false == \text{“}fall\text{”})$ $\quad B.code = B_1.code \| B_2.code \| label(B_1.false)$ $else$ $\quad B.code = B_1.code \| B_2.code$
$B \rightarrow B_1 \| B_2$	$if(B.true == \text{“}fall\text{”})$ $\quad B_1.true = newlabel()$ $else$ $\quad B_1.true = B.true$ $B_1.false = \text{“}fall\text{”}$ $B_2.true = B.true$ $B_2.false = B.false$ $if(B.true == \text{“}fall\text{”})$ $\quad B.code = B_1.code \| B_2.code \| label(B_1.true)$ $else$ $\quad B.code = B_1.code \| B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$if(B.true != \text{“}fall\text{”} \&\& B.false != \text{“}fall\text{”})$ $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('if' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.true)$ $\quad\quad \| gen('goto' B.false)$ $elif(B.true != \text{“}fall\text{”} \&\& B.false == \text{“}fall\text{”})$ $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('if' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.true)$ $elif(B.true == \text{“}fall\text{”} \&\& B.false != \text{“}fall\text{”})$ $\quad B.code = E1.code \| E2.code$ $\quad\quad \| gen('ifFalse' E_1.addr$ **rel**$.op\ E_2.addr 'goto' B.false)$ $else$ $\quad$ error
$B \rightarrow true$	$if(B.true != \text{“}fall\text{”})$ $\quad B.code = gen('goto' B.true)$ $else$ $\quad B.code = ''$
$B \rightarrow false$	$if(B.false != \text{“}fall\text{”})$ $\quad B.code = gen('goto' B.false)$ $else$ $\quad B.code = ''$
$B \rightarrow$ **id**	$if(B.true != \text{“}fall\text{”} \&\& B.false != \text{“}fall\text{”})$ $\quad B.code = gen('if' top.get(id.lexeme)' == 1 goto' B.true) \| gen('goto' B.false)$ $elif(B.true != \text{“}fall\text{”} \&\& B.false == \text{“}fall\text{”})$ $\quad B.code = gen('if' top.get(id.lexeme)' == 1 goto' B.true)$ $elif(B.true == \text{“}fall\text{”} \&\& B.false != \text{“}fall\text{”})$

	$B.code = gen('ifFalse' top.get(id.lexeme)' == 1 goto' B.false)$ *else* error

Code intermédiaire pour les statements

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