TD de Génération de code pour la machine virtuelle

On s'intéresse à la génération de code pour la machine abstraite utilisée en projet. On suppose qu'on ne manipule que des variables dont l'adresse en mémoire est à un décalage connu par rapport au fond de pile (représenté par le registre `GP` de la machine) . On manipule donc une variable via les instructions `PUSHG` et `STOREG` en fournissant le décalage associé à cette variable. On suppose qu'on a défini dans notre compilateur une fonction `adresse` qui renvoie le décalage associé à toute variable du programme. On donne ci-dessous les attributs pour engendrer du code pour des expressions arithmétiques et l'affectation dans ce cadre très simple. L'attribut `code` est un attribut synthétisé qui contient le texte des instructions produites (les lignes produites dans l'attribut `code` sont implicitement concaténées et séparées par un passage à la ligne pour être conforme à du code de la machine abstraite avec une instruction par ligne) :

```
E ::= Id
   code(E) := PUSHG adresse(Id.lexval)

E::= Cste
   code(E) := PUSHI Cste.lexval

E ::= E + E                          E ::= E * E
   code(E.0) := code(E.1)               code(E.0) := code(E.1)
                code(E.2)                            code(E.2)
                ADD                                  MUL
```

Comme en C, l'affectation ci-dessous est une expression qui retourne la valeur de sa partie droite :

```
E::= Id = E     code(E.0) :=  code(E.1)
                              DUPN 1          -- voir ci-dessous
                              STOREG adresse(Id.lexval)
```

Si on applique ce mécanisme à l'instruction `x := y + z * 2` dans un contexte où la fonction `adresse` associe les décalages 1 à `x`, 0 à `y` et 2 à `z`, le code engendré[1] sera

```
PUSHG 0          -- empiler la  valeur de  y
PUSHG 2          -- empiler la valeur de  z
PUSHI 2          -- empiler la constante  2
MUL              -- calculer z  *  2
ADD              -- ajouter y au résultat précédent
DUPN 1           -- duplique le résultat, un exemplaire sert pour l'instruction suivante
STOREG 1         -- stocker le résultat dans l'emplacement associé à  x
```

Le résultat de l'exécution de `y + z * 2`, et donc de l'affectation, est donc laissé en sommet de pile grâce au `DUPN 1` qui permet de conserver une copie de la valeur en sommet de pile.

Exercice 1 : On considère une sous-grammaire partielle décrivant la boucle `for` du langage C :

```
Instruction ::= for ( EOpt ; EOpt ; EOpt ) Instruction
Instruction ::= EOpt ;
Instruction ::= { InstructionsLOpt }
InstructionsLOpt::= ε
   | Instruction InstructionsLOpt
EOpt :: = ε | E
```

1 On suppose que cette instruction se situe dans un contexte où les variables ont été bien déclarées , ce qui aura pu être vérifié lors des vérifications contextuelles. On ne produit du code que pour des programmes corrects !

```
E ::= Id = E
    |  E , E
    | … expressions vues en cours et en TD. Voir les attributs en début d'énoncé …
```

La règle `E::= E , E` correspond à l'opérateur `','` du langage C qui évalue la sous-expression gauche, en ignore le résultat puis évalue la sous-expression droite et renvoie la valeur de cette dernière. La partie « initialisation » de la boucle `for` peut être vide ; sa partie « condition de sortie » aussi, auquel cas elle est considérée comme « true » ; enfin, la partie « incrémentation » peut aussi être omise.

Exemples : `for(;;) ;` -- boucle infinie avec un corps vide !
`for(x=0, y=z-12; x > z; x=x+1) x=y;`

1. Donnez des attributs pour engendrer du code pour les règles de la grammaire ci-dessus. Il est inutile de redonner les règles pour les expressions arithmétiques vues en cours. On dispose de la fonction `adresse` évoquée ci-dessus ainsi que d'une fonction n`ewLabel` qui permet **à la compilation** d'engendrer des étiquettes pour les instructions du code (adresses de sauts)

2. Donnez le code produit par votre système d'attributs pour les deux exemples ci-dessus. Pour la fonction `adresse` on utilisera les décalages de l'exemple de la page précédente.

3. On considère en plus les deux instructions `continue` et `break` qui permettent respectivement dans le corps de la boucle d'aller directement à la prochaine itération de boucle (après avoir fait la partie « incréments » si elle existe) et à quitter directement la boucle.

Indiquez les vérifications contextuelles spécifiques à mettre en œuvre pour ces deux instructions et comment vous modifiez votre réponse à la Question 1 pour prendre en compte ces deux nouvelles instructions. Décrivez les attributs supplémentaires qu'il faut introduire, leur nature synthétisée ou héritée et le principe de leur calcul. Les règles de calcul ne sont **pas** demandées.

Exercice 2 : On considère l'instruction Ada de boucle généralisée dont la structure est la suivante :

```
loop C1
     when E1 do C'1 exit;
     C2
     …
     when En do C'n exit;
     Cn+1
end loop;
```

où chaque C_1, C'_1, …, C'_n, C_{n+1} est une liste d'instructions quelconques, éventuellement vide et E_1, … E_n des expressions. Le fonctionnement de cette boucle consiste à en exécuter le corps, i.e. les C_i et les E_i séquentiellement (et de façon répétitive) jusqu'à ce que l'une des expressions E_i soit satisfaite, auquel cas on exécute C'_i et on quitte la boucle. S'il n'y a pas de clause `when … exit`, ou si les expressions s'évaluent toujours à *`false`*, la boucle est donc répétée indéfiniment.

1. Donnez une grammaire pour cette construction. On notera, sans les détailler, `E` une expression quelconque et `I` une instruction quelconque différente de la boucle ci-dessus. **On tiendra compte du fait que de telles boucles sont des instructions comme les autres et peuvent être emboîtées.**
2. On considère le code intermédiaire pour la machine virtuelle du projet ; les attributs vus en cours qui n'ont pas à être redonnés. La fonction `newLabel()` produit des noms

d’étiquettes différentes à chaque appel lors de l’application de votre schéma de génération de code à une occurrence de production.

Donnez un système d’attributs comportant des attributs synthétisés et des attributs hérités pour engendrer du code pour cette boucle. Expliquez comment vous transmettez les étiquettes nécessaires aux instructions `JZ` et `JMP` pour gérer les différents enchaînements dans le code. On pourra réutiliser directement les attributs du cours pour la génération de code des expressions et l'affectation (en tenant compte du fait qu'**en Ada l'affectation est une instruction, pas une expression**. Son exécution ne renvoie donc pas de valeur).

3. Donnez le code engendré pour le programme source ci-dessous, en supposant que les variables `x`, `y` et `z` sont à des déplacements de 0, 1 et 2 par rapport à `GP` :

```
loop
   x := x + 1;
   when x > y do y := y + 1 exit;
   loop
      y := y + 2;
      when y > 1 do exit;
      y := y + z;
   end loop;
   when x > z do exit;
end loop;
```