TP 5 : Compiler un langage rudimentaire vers de l'assembleur MIPS32

Dans ce TP, nous allons considérer un fragment minime des langages étudiés jusqu'à présent, et écrire un visiteur de son AST qui produira du code assembleur.

Pour le code assembleur, vous pourrez vous rapporter à la documentation : `https://ecs-network.serv.pacific.edu/ecpe-170/tutorials/mips-instruction-set`.

Pour ce TP, on considérera un langage élémentaire permettant de calculer des expressions arithmétiques, avec lectures, impressions et affectations. Un code vous est fourni, mais vous pouvez également réutiliser vos travaux des TP précédents, en ne travaillant que sur une partie du langage.

Le langage

Le langage vise à produire du code assembleur, sans considérer les optimisations (*Generation of Assembly with Zero Optimisation* : `gazo`). Il contient les éléments suivants :

- Expressions :
  - Entiers
  - Opérations habituelles unaires et binaires sur les entiers.
  - Variables
  - Entier lu sur l'entrée standard.
- Instructions :
  - Affectations
  - Impressions
  - Suites d'instructions

L'AST comporte donc une classe pour chacun des points ci-dessus, ainsi qu'un nœud `Program` contenant l'instruction à exécuter.

Analyse sémantique

Pour ce langage, toutes les expressions sont de type entier, donc une analyse de typage n'est pas nécessaire. Une fois l'AST construit, le seul visiteur à implémenter est celui qui génèrera le code assembleur.

Génération de code

Vous devez écrire un visiteur de l'AST, qui produit une chaîne de caractères correspondant au code assembleur. C'est le fichier `mips/Translate.java`, qui hérite du visiteur générique.

La difficulté est qu'en assembleur, il n'y a pas vraiment d'expressions (seulement les entiers littéraux, les noms de registres, les étiquettes d'instructions). Il faudra donc faire en sorte, lors de la visite des expressinos du langage `gazo` de produire des **instructions assembleur** qui effectuent le calcul de ces expressions.

Calcul des expressions et des instructions – principe de base

On pourra utiliser le registre `$v0`, et faire en sorte que chaque expression `e` en `gazo` soit traduite en un code qui stocke le résultat dans `$v0`. Par exemple, la traduction d'une expression correspondant à un entier littéral n (`ExpInt`) sera compilée vers la chaîne de caractères "`li $v0 n`". Pour les expressions composées, le code généré devra passer par un petit algorithme (voir plus bas) et pourra être composé de beaucoup d'instructions. Une expression pourra donc être traduite par un code contenant un grand nombre de lignes, tant qu'à la fin il stocke le résultat dans `$v0`.

Grâce au mécanisme décrit ci-dessus, on pourra compiler également les instructions. Par exemple, `print(e);` sera décomposée en deux étapes :

1. Générer le code correspondant à `e`
2. Générer le code correspondant à une impression d'entier en assembleur, en supposant que celui qu'on cherche est stocké en `$v0` à ce moment-là.

Une difficulté est la gestion des variables, notamment pour les affectations. Plusieurs solutions sont possibles : utiliser les registres, utiliser la mémoire (le tas), ou bien les deux. Implémentez d'abord une solution complète et fonctionnelle uniquement avec les registres, avant d'améliorer le compilateur en intégrant la mémoire.

Utilisation des registres MIPS32

Dans un premier temps, nous allons considérer que l'exécution du code est possible en utilisant uniquement les registres de l'assembleur, et pas la pile ni le tas. Cela contraint naturellement le nombre de variables que peut définir l'utilisateur; ce problème sera considéré plus tard dans le TP. On pourra utiliser les registres `$t0` à `$t9` pour les variables, et `$s0...$s9` pour les calculs intermédiaires.

Le visiteur devra donc contenir des informations globales pour faire le lien entre les variables du langage et les registres. En particulier, il devra contenir les mécanismes suivants :

- Indiquer combien de registres sont utilisés par les variables

- Retourner un nom de nouveau registre (pour l'écriture dans une nouvelle variable) s'il en reste de disponible. **Dans cette solution, l'utilisateur ne peut pas utiliser plus de 10 variables différentes**
- Retourner le registre associé à une variable (pour la lecture du contenu d'une variable).

Les variables du programme source seront donc d'abord traitées de la façon suivante :

- Lors de l'affectation d'une expression à une variable :
  - L'expression est compilée dans un code `c` (stockant donc le résultat du calcul dans `$v0`).
  - Si la variable n'a jamais été affectée :
    - le programme vérifie qu'il reste des registres disponibles. (sinon il provoque une erreur et quitte la génération de code)
    - il associe un nouveau registre à cette variable
  - Sinon : il récupère le registre qui a été associé à la variable auparavant.
  - Il produit le code `c'` qui charge le résultat dans le registre adapté.
  - Il retourne `c` suivi de `c'`
- Lors de l'accès au contenu d'une variable : il récupère le registre associé à la variable, et charge son contenu. Si la variable n'a aucun registre associé, une erreur est signalée.

L'association entre les variables est les registres pourra être gérée à travers une `Map` unique.

Expressions composées

Pour calculer une expression unaire `Op e`, c'est assez simple :

- Produire le code `c` donnant le calcul de `e` dans `$v0`.
- Produire l'instruction faisant le calcul adapté à l'opération. (c'est-à-dire pour une soustraction par exemple, `v0 <- 0 - v0`)

Lors du calcul d'une expression binaire `eg Op ed`, une solution est de procéder comme suit :

- Produire le code calculant l'évaluation de `eg`, (on supposera donc que le résultat est chargé dans `$v0` à la fin de ce programme.)
- Utiliser un registre temporaire inutilisé (par exemple `$s0`) pour stocker le résultat
- Faire de même pour `eg`, et stocker le résultat dans `$s1`.

- Charger dans `$v0` le résultat de l'opération nécessaire (l'équivalent de `Op` en assembleur) appliquée à `$s0` et `$s1` dans `$v0`

Écrivez les méthodes de visite pour les autres expressions, sachant que chaque visite produit du code à l'issue duquel la valeur calculée par la visite de chaque expression est toujours chargé dans `$v0`.

Pour la visite des instructions, le code assembleur à produire se déduit également de ce qui précède.

Premiers tests

Commencez par tester votre compilateur sur `test.gazo`. Produisez ensuite un petit jeu de tests avec des variables, lectures, etc...

L'exécution du main produit le code assembleur. Enregistrez le (avec redirection sortante >) dans un fichier `test.s`, et exécutez-le dans le simulateur SPIM.

Tests suivants

Modifiez le fichier `test.gazo` en changeant le parenthésage de l'expression (remplacez (1+5)+5 par 1+(5+5)).

Que se passe-t-il ?

Analysez le code assembleur produit et tâchez d'expliquer le comportement du programme en suivant l'exécution et l'évolution des registres avec précision.

Modifiez le programme pour obtenir le comportement souhaité ? (indice : pile).

Meilleure utilisation de la mémoire

Contraindre l'utilisateur à n'utiliser que dix variables n'est pas très raisonnable. Modifiez votre programme pour que les variables puissent correspondra à des adresses en mémoire et non plus à des registres. (on notera que la commande `label : .space 4` réserve 4 octets en mémoire à l'adresse indiquée par `label`).

Le compilateur modifié devra allouer un registre à une variable si un registre est disponible, et sinon (si plus de dix registres ont été affectés par exemple, et que `$t0...$t9` ne sont donc plus disponibles), il lui réservera une place de 4 octets.

Pour aller plus loin

Pour se rapprocher du comportement efficace d'un compilateur, il faudrait également être en mesure de libérer des registres affectés par des variables, quand ils ne sont plus utilisés. En effet, les accès *via* des registres sont plus efficaces que les accès à la mémoire, il est donc de mise de les utiliser autant que possible pour optimiser l'efficacité du code.

Pour cela, la solution est de procéder à une *analyse de vie* des variables, et d'appliquer un algorithme de coloration de graphes. Cet aspect sera brièvement abordé en cours, mais n'est pas au programme pour l'instant. Si toutefois vous souhaitez approfondir ce point, contactez l'enseignant pour des indications et/ou des références.