Projet de Compilation – Troisième et dernier devoir

Génération et optimisation de code

Le but de cette étape est de poursuivre le projet précédent jusqu'à la production de code intermédiaire à 3 adresses présenté en annexe B. Nous ne proposons pas de produire du code pour une machine réelle ni même pour une machine virtuelle car cela demanderait un travail supplémentaire technique assez long et fastidieux.

Pour cet exercice, nous considérons que la machine cible de ce code à 3 adresses

- A un nombre infini de registres
- A une mémoire infinie adressable par $M[k]$ où k est un entier
- Permet des branchements inconditionnels et conditionnels non limités dans l'espace de la mémoire
- A un mot pour chacun de ses registres de la taille de l'entier et du pointeur du langage source.

Méthodes

Nous retenons deux méthodes et laissons à votre discrétion le choix de ces méthodes.

1. La première consiste à faire une représentation intermédiaire sous forme d'un arbre pour chaque expression et pour chaque instruction suivant le livre d'A. Appel. C'est à partir de cet arbre que l'on produit le code linéaire à 3 adresses.
2. On produit directement le code à 3 adresses à partir de l'arbre de syntaxe abstraite comme proposé par le livre de Aho, Sethi, Ulman.

1 Première méthode

Voir le cours pour une explication des différentes classes. Nous avons volontairement simplifié `JUMP` dans ce qui suit.

- Expressions
  - CONST(int value)
  - NAME(Label label)
  - TEMP(Temp temp)
  - BINOP(int binop, Exp left, Exp right)
  - MEM(Exp exp)
  - CALL(Exp func, ExpList args)
  - ESEQ(Stm stm, Exp exp)
- Instructions
  - MOVE(Exp dst, Exp src)
  - EXP(Exp exp)

— JUMP(Label label)
— CJUMP(int binop, Exp left, Exp right, Label iftrue, Label iffalse)
— SEQ(Stm first, Stm rest)
— LABEL(Label label)

1.1 buildIntermTree()

Les objets de l'arbre de syntaxe abstraite envoient un message `buildIntermTree` permettant de créer les instances de code intermédiaire.

1.2 Linéarisation

L'arbre ainsi défini est aisé à produire, cependant il ne correspond pas immédiatement à du code linéaire à 3 adresses :

— CJUMP fait intervenir deux labels, alors que le code à 3 adresses ne permet qu'un saut par instruction.
Solution : Faire suivre immédiatement le CJUMP du Label iifalse
— Une expression pouvant contenir une instruction (par l'intermédiaire d'ESEQ), il se peut que l'instruction ait un effet sur cette dernière (modification d'un temporaire par exemple). La règle est donc dans un ESEQ de produire le code de l'instruction avant celui de l'expression.
Solution : Réécrire systématiquement les expressions contenant des ESEQ pour les faire « monter ».
— Les appels à fonction CALL peuvent être imbriqués directement dans les expressions et instructions.
Solution : Produire systématiquement une affectation vers un temporaire pour tout appel de fonction.

A l'issu de ces procédures, on obtient un code intermédiaire immédiatement traduisible en code à 3 adresses.

1.3 build3AddressCode()

Produire le code à 3 adresses directement à partir des arbres de code intermédiaire.

2 Seconde méthode

Le principe ici est de produire directement du code à 3 adresses à partir de la représentation de la syntaxe abstraite.

Le code à 3 adresses devant être manipulé plusieurs fois, il convient de ne pas l'implémenter immédiatement par production de texte, mais par une classe `ThreeAddressCode()`

2.1 build3AddressCode()

Pour chaque expression et instruction de l'arbre de syntaxe abstraite, produire le code à 3 adresses linéaire :

— Linéarisation des structures de contrôle
— Linéarisation des expressions

3 Production de code complexe

Chaque équipe devra proposer l'implémentation au choix
- Adressage des éléments de tableaux
- Adressage des champs d'une structure (ou des propriétés et méthodes des classes)

3.1 Adressage des éléments de tableaux

L'adressage d'un élément de tableau $T[i]$ revient à $base + (i - binf) \times t$ où $base$ est l'adresse du tableau, $binf$ la borne inférieure des indices du tableau et t la taille de chaque élément du tableau.

Il vient que l'adressage d'un élément $T[i, j]$ d'un tableau à deux dimensions est $base + ((i - binf_1) \times n_j + j - binf_2) \times t$ où $binf_1$ et $binf_2$ sont les bornes inférieures des indices i, j et n_j est le nombre valeurs que peut prendre j.

En retenant que notre langage ne permet que l'allocation statique des tableaux, produire le code à 3 adresses pour l'adressage des tableaux en général.

3.2 Adressage des éléments d'une structure

Les expressions de type complexes permettent de représenter les structures et pour chaque structure, l'ensemble des champs qu'elle contient. Les champs correspondent à des éléments typés et dont l'emplacement relatif est fonction des autres champs qui précèdent dans la même structure.

L'expression $o.name_i$ où o est un objet instance d'une structure $struct\{name_1 : type_1; name_2 : type_3; \ldots name_i : type_i; \ldots\}$ correspond à un adressage $o[k]$ où k est la somme de la taille des types $type_1, type_2, \ldots type_{i-1}$ de 1 à $i - 1$

En retenant que notre langage ne permet que le typage statique, produire le code à 3 adresses pour l'adressage des éléments d'une structure en général. L'extension vers les objets d'un langage de classe se fera sans prendre en considération les espaces de noms ni les phénomènes d'encapsulation (en gros tout est déclaré systématiquement « public »).

Annexe B : Code à trois adresses

- $x = y$ `op` z
  Instruction d'affectation où *op* est un opérateur binaire arithmétique ou logique
- $x =$ `op` y
  Instruction d'affectation où *op* est un opérateur unaire arithmétique, logique, de décalage et de conversion
- $x = y$
  Instruction de copie où la valeur de y est affectée à x
- *Label* L
  Etiquette l'instruction qui suit par le label L
- `JUMP` L
  Branchement inconditionnel qui a pour effet de faire exécuter ensuite l'instruction étiquetée par L
- `IF` x *op* y `JUMP` L
  Branchement conditionnel qui a pour effet de faire exécuter ensuite l'instruction étiquetée par L si la relation x *op* y est satisfaite. Où *op* est un opérateur relationnel ($<$, $>$, $\leq$, ...)
- Instructions d'appel de procédures.
  - `PARAM` x
    Passe un paramètre par valeur à une procédure
  - `CALL` p, n
    Appel de la procédure p qui prendra en compte n paramètres
  - `RETURN` y
    La procédure renvoie la valeur y
- Les instructions d'affectation de variables
  - $x = Mem[i]$
    Où $Mem[i]$ désigne l'emplacement situé à l'adresse i en mémoire
  - $Mem[i] = y$
- Les instructions d'affectation d'adresses et pointeurs
  - $x = \&y$
    Où $\&y$ désigne l'adresse de y
  - $x = *y$
    Où $*y$ désigne la valeur pointée par y
  - $*x = y$