Examen de compilation

M1 informatique

20 juin 2023

Documents : interdits

Durée : 2h

Barème : donné à titre indicatif

Enseignant : Jules Chouquet

Exercice 1 : Questions de cours

question 1 (/7) : Décrivez précisément les différentes phases d'un compilateur dans l'ordre de leur exécution, en identifiant leurs responsabilités respectives. (Les phases n'ayant pas été abordées en cours sont facultatives.)

question 2 (/1) : Expliquez en quelques mots la différence entre un langage interprété et un langage compilé. Citez au moins un exemple de chaque. Précisez le cas de Java.

question 3 (/1) : Pourquoi les *expressions* du code source à compiler ne peuvent pas être toujours traduites en simples expressions dans la représentation intermédiaire ? Donnez un exemple.

Exercice 2 : Table des symboles

Considérez le programme suivant :

```
1          int f(int x){
             int y = 2*x;
3              while(x<y){
                 float x;
5                y = y/2;
                 x = y;
7              }
```

```
        return y+x;
9       }

11      int g(int y){
          print(f(y+1));
13      }
```

question 1 (/0,5) : Pour chaque utilisation de variable (hors déclaration), indiquez la ligne à laquelle se situe la déclaration qui lui est associée[1].

question 2 (/2) : Représentez de façon schématique et précise la table des symboles construites à l'issue de l'analyse du programme; avec l'algorithme vu en cours.

question 3 (/1,5) : Jacques propose de construire la table des symboles en modifiant le comportement de l'enregistrement des variables, son idée est de procéder ainsi : "On a une seule grande table, ou chaque variable est associée à une liste chaînée, et dès qu'une déclaration est faite, on rajoute le type déclaré à la fin de la liste. Comme ça la dernière déclaration est toujours accessible en fin de liste."

Commentez la proposition de Jacques en la comparant avec celle vue en cours. Illustrez cette comparaison par un exemple où les deux approches aboutissent à un résultat différent.

question 4 (/2) : Donnez (sous la forme que vous souhaitez (algorithme, visiteur,...), la fonction de vérification de typage pour l'affectation d'une expression à une variable. On considèrera que la table des symboles et ses fonctions, ainsi que la vérification de typage pour les autres constructions syntaxiques sont déjà implémentées.

Exercice 3 : Génération

question 1 (/2) : Considérons l'algorithme suivant pour la génération de code assembleur associé à une expression binaire de la représentation intermédiaire. On suppose que la fonction `opToAsm` retourne l'instruction assembleur associée à une opération binaire arithmétique (par exemple `opToAsm(PLUS)` renverra `add`). On suppose que le code généré pour les sous expressions stocke toujours le résultat dans le registre $v0.

```
1   generate(ExpBin e){
        resL = generate(e.leftExp())
3       resG = generate(e.rightExp())
        op = opToAsm(e.getOp())
5
```

[1] vous pouvez donner ces associations sur le sujet, précisez-le sur votre copie le cas échéant.

```
        retourner :
7               resL
                + ''\n move $t1 $v0''
9               + resR
                + ''\n move $t2 $v0''
11              + ''\n op $v0 $t1 $t2''
}
```

Pourquoi cet algorithme n'est-il pas satisfaisant ? Donnez un exemple d'expression pour laquelle le résultat calculé est différent du résultat attendu, en justifiant votre réponse.

question 2 (/3) : Traduisez le programme suivant en code intermédiaire[2], puis en assembleur[3].

```
int f(int x, int y){
2       int z = 0;
        while(x<y){
4               z++;
                y--;
6       }
        return z;
8   }
```

[2] vous pouvez utiliser une représentation moins formelle que l'implémentation vue en cours, notamment pour les frames

[3] la documentation ne vous étant pas fournie, vous pouvez utiliser du code pseudo-assembleur, dont le jeu d'instructions doit être comparable à celui des machines MIPS32.