Compilation – Projet

1 Présentation

Le but de ce projet est d'implémenter un compilateur, qui traduit le langage TCL (Théorie de la Compilation et des Langages) en langage assembleur, dont la syntaxe a été vue en cours de structure des ordinateurs.

1.1 Modalités pratiques

Le projet est décomposé en trois parties, correspondant à trois étapes de la compilation :

1. inférence de types
2. génération de code
3. allocation de registres

Chaque groupe de TD est décomposé en deux groupes de 9-10 personnes. Chacun de ces sous-groupes devra réaliser un compilateur complet, en se décomposant en trois sous-sous-groupes qui s'occuperont chacun d'une partie. Il y aura donc au total 12 groupes, pour 4 compilateurs.

La date de rendu est le **28 janvier à 23h59**. Vous devez envoyer avant cette date un mail aux deux intervenants `theo.pierron@univ-lyon1.fr` et `mathieu.lefort@univ-lyon1.fr`, contenant les sources de votre projet, ainsi que tout document qui vous paraîtra utile (programmes de test, descriptif, mode d'emploi,...). **N'envoyez pas de fichiers compilés/exécutables, au risque de voir votre mail traité comme spam !**

Une soutenance aura lieu (sauf indication contraire) le 29/01 après-midi pour les deux compilateurs du groupe 1, et le 30/01 matin pour les deux compilateurs du groupe 2 (horaires à préciser).

1.2 Le projet

Un squelette du code est disponible sur la page du cours. Cette base à compléter est une proposition d'architecture, que vous avez le droit de modifier. Elle contient :

- `GrammarTCL.g4` la grammaire du langage source TCL
- `GrammarTCL*.java` le code généré par ANTLR.
- `simproc.py` le simulateur de code assembleur.
- `simcode.py` le simulateur de code assembleur avec un nombre de registres non-borné.
- `input` le fichier contenant le code TCL source.
- `prog.asm` le fichier contenant le code assembleur obtenu.
- `entrees.txt` et `sorties.txt` les fichiers nécessaires au fonctionnement du simulateur assembleur.
- D'autres fichiers détaillés dans les sections qui les concernent.

La grammaire du langage source est fournie dans le fichier `GrammarTCL.g4`. Un programme est composé d'une suite de déclarations de fonctions, la dernière étant la fonction `main`. Chaque fonction est composée d'une suite d'instructions, la dernière étant nécessairement un `return`.

Quelques spécifications du langage :

- Le mot clé `auto` permet de ne pas expliciter le type d'une variable, d'un argument de fonction, ou de retour d'une fonction.
- L'affectation `t[x] = y` change la longueur du tableau `t` si nécessaire. Autrement dit, si `t` a taille 3 avant l'instruction, et `x` vaut 5, alors `t` a taille 6 après l'instruction.
- L'instruction `print(x);` affiche x si c'est un entier ou un booléen. Si c'est un tableau, on veut afficher $[x[0], x[1]...]$.
- La redéclaration de variables/de fonctions ne sont pas autorisés. Ainsi, deux fonctions ne peuvent pas avoir le même nom (même si leurs signatures sont différentes), et les codes suivants doivent lever une erreur à la compilation.

```
int x = 1;
int x = 0;
```

```
int f(int x) {
    int x = 0;
    return x;
}
```

En cas de doute sur les spécifications du langage, les spécifications du langage C font foi. Si le doute persiste, envoyez-nous un mail.

2 Inférence de types (3-4 personnes)

Le but de cette section est d'analyser le code source pour :

- vérifier que le typage est correct (ou lever des erreurs si ce n'est pas le cas)
- inférer le type (le plus général possible) des éléments déclarés en `auto`

Les types utilisés dans ce projet sont ceux reconnus par le non-terminal `type` de la grammaire. Il s'agit donc d'entiers, de booléens, ou de tableaux d'un certain type.

Les sources contiennent un sous-dossier `Type` qui contient des classes représentant les différents types. Ces classes héritent toutes de la classe abstraite `Type.java`, et la plupart des méthodes doivent être implémentées. Vous devez ensuite implémenter les méthodes du fichier `TyperVisitor.java`, qui effectuent l'inférence de type en visitant les noeuds de l'AST. À la fin de la visite, si aucune erreur de typage n'a été détectée, l'attribut `types` devra contenir le type de chaque variable du programme. C'est cette information qui doit être fournie au deuxième groupe.

Quelques spécifications supplémentaires:

- Tous les `return` d'une fonction doivent avoir des types compatibles, ainsi le code suivant doit lever une erreur de typage:

```
auto f () {
    if (0 == 1) {
        return true;
    } else {
        return 0;
    }
    return {x};
}
```

- L'appel de fonction ne force pas son type. Par exemple :

```
auto f(auto x){
    return {x};
}

int main(){
    int[] x = f(0);
    bool[] y = f(true);
    return 0;
}
```

  est un programme valide et la fonction `f` a pour type `A -> tab[A]` à tout moment de la compilation.

3 Génération de code (2-3 personnes)

Le sous-dossier `Asm` contient les classes permettant de représenter un code assembleur (comme au TP2). Le but de cette section est de générer un objet `Program` à partir de l'AST et des types obtenus par le groupe 1, en implémentant les méthodes de `CodeGenerator.java`. Cette classe contient des visiteurs de l'AST qui renvoient un objet de type Program, contenant un *code linéaire*, c'est-à-dire un code assembleur qui utilise un nombre arbitraire de registres. Vous pouvez simuler l'exécution de ce code en utilisant le simulateur `simcode.py`.

Quelques spécifications:

- Les booléens seront représentés par les entiers 0 (False) et 1 (True).
- La gestion des tableaux (allocation de mémoire et initialisation) seront effectuées comme décrit en CM.

4 Allocation de registres (2-3 personnes)

Le sous-dossier `Graph` contient des classes permettant de représenter des graphes dirigés et non dirigés.

Le but de cette partie est de transformer le code obtenu par le groupe 2 en un vrai code assembleur (sur 32 registres), qui doit pouvoir être simulé par `simproc.py`. Plusieurs approches (plus ou moins naïves) sont possibles. Nous vous demandons d'implémenter l'analyse vue en CM, à savoir :

- générer le graphe de contrôle
- calculer les ensembles LV_{entry} et LV_{exit}
- générer le graphe de conflits
- le colorer (la classe `UnorientedGraph` contient déjà une méthode de coloration)
- modifier le code fourni par le groupe 2 en conséquence. Il faut notamment penser à ce qui doit se passer lorsque le graphe n'est pas 32-colorable !