CMP1 – Construction des compilateurs
TYLA – Typologie des Langages

EPITA – Promo 2011
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices et ordinateurs interdits.

Janvier 2008 (1h30)

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante. Une lecture préalable du sujet est recommandée.

Merci de répondre sur votre copie, même pour les questions à choix multiples.

1 Passage d'arguments

1. Quel(s) type(s) de passage(s) d'argument(s) est (sont) utilisé(s) en C ?
   (a) passage par valeur/copie (*call by value*)
   (b) passage par référence (*call by reference*)
   (c) passage par résultat (*call by result*)
   (d) passage par nom (*call by name*)
2. Même question avec C++ (attention, il y a un piège !).
3. Même question avec Tiger.
4. Il n'y a pas de type référence en Tiger ; comment peut-on les simuler avec des constructions du langage existantes ?

2 Const

À la différence de C++, Tiger est dépourvu de mot clef `const`. On se propose dans cet exercice de le rajouter dans le langage, en passant notamment en revue les différentes parties de notre compilateur afin de déterminer quels sont les aménagements nécessaires.

Rappel : le langage Tiger utilisé dans cet exercice est celui qui est décrit dans le Manuel de Référence du Compilateur Tiger, qui est utilisé comme référence tout au long du projet du même nom.

2.1 Préliminaires

1. À quoi sert `const` en C++ ? Il y a plusieurs réponses possibles, une seule est attendue).
2. Parmi les lignes suivantes, quelles sont celles qui sont valides en C++ ? Justifiez votre réponse.
   (a) `int i = 42; const int* const p = &i;`
   (b) `int i = 42; const int const * p = &i;`

(c) `int i = 42; int const * p = &i;`

(d) `int i = 42; int const * const p = &i;`

3. Même question.

(a) `int i = 42; const int& const p = i;`

(b) `int i = 42; const int const & p = i;`

(c) `int i = 42; int const & p = i;`

(d) `int i = 42; int const & const p = i;`

4. Soit la classe suivante :

```
struct C
{
  void m1 () {}       // (0).
  void m1 () const {} // (1).
  void m2 () {}       // (2).
  void m3 () const {} // (3).
};
```

Pour chacun des cas suivants, indiquer quelle méthode est appelée (0, 1, 2 ou 3) ou s'il y a une erreur de compilation, le cas échéant.

(a) `C o; o.m1();`

(b) `const C o = C(); o.m1();`

(c) `const C o = C(); o.m2();`

(d) `C o; o.m3();`

(e) `void (C::*m)() const = &C::m1; (C().*m)();`

(f) `void (C::*m)() = &C::m3; (C().*m)();`

2.2 Partie frontale (*front end*)

Cette partie s'intéresse aux modifications à apporter au front end Tiger pour prendre en compte `const`. Nous ne considérerons ce nouveau mot-clef que comme qualificatif de types (le cas des méthodes « constantes » n'est pas à traiter).

1. **Spécification lexicales.** Que faut-il ajouter aux spécifications lexicales du langage pour supporter `const` ?
2. **Scanner.** Comment étendre le scanner, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`scantiger.ll`' pour supporter ces nouvelles spécifications lexicales ?
3. **Spécification syntaxiques.** Que faut-il ajouter aux spécifications syntaxiques du langage pour supporter `const` ?
4. **Syntaxe abstraite.** Faut-il effectuer des modifications ou des ajouts dans la syntaxe abstraite du langage pour supporter `const` ? Si oui, lesquelles ?
5. **Parser.** Comment étendre le parser, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`parsetiger.yy`' pour supporter ces nouvelles spécifications syntaxiques ?
6. **Liaison des noms.** Que faut-il changer dans le `Binder` vis-à-vis de `const` ?
7. **Vérification des types.** Qu'allez-vous changer dans le module `type` du compilateur ?
8. **Représentation intermédiaire** Faut-il ajouter/modifier quelque chose dans le langage Tree ?
9. **Traduction (vers IR).** Y'a-t-il des changements à apporter au visiteur chargé de la traduction vers la représentation intermédiaire ? Justifiez votre réponse.

2.3 Partie terminale (*back end*)

Nous n'avons pas étudié le back end de `tc` à ce stade de l'année, mais nous avons déjà un évoqué ses composants, et vous avez déjà été sensibilisés à des notions connexes dans d'autres cours (assembleur, architecture des ordinateurs, etc.).

Grossièrement, le back end de `tc` effectue deux tâches :

L'allocation de registres Chaque fonction peut contenir un nombre arbitraire de variables ou *temporaires* ; or un processeur n'a qu'un nombre fini de *registres*. L'allocateur de registres a pour tâche de résoudre ce problème en faisant appel à la mémoire au besoin.

La génération de code en langage d'assemblage Il s'agit de la dernière étape de traduction, qui fait le pont entre la représentation intermédiaire (en Tree) et la sortie (en langage d'assemblage MIPS ou IA-32 dans notre cas).

1. **Allocation de registres.** Est-ce que l'adjonction de `const` à Tiger modifie l'allocateur de registres ?
2. **Génération de code.** De même, la génération de code à partir de la représentation intermédiaire est-elle affectée ?