COA

Pablo De Oliveira

Table des matières

Retards Attention : un train d'étudiants en retard à 5 minutes d'intervalle fait perdre 20 minutes au cours ! Une petite tolérance est certes normale, mais cela ne sera en aucun cas systématique (revenez à la pause).

1 Introduction

Pourquoi faire de la compilation dans un master de HPC ? Depuis quelques années, les architectures deviennent de plus en plus complexes (pipeline, machines vectorielle, multi-noeud, interfaces réseaux haute performance, etc).

Dans les machines actuelles, chaque instruction est codée comme un nombre binaire, qui a un sens pour la machine. Pour des raisons évidentes de compréhension, on préférera utiliser des *mnémoniques* du type `mov $r0, $r1` au lieu de 10001001 11 000 011. L'*assembleur* est le programme convertissant les mnémoniques en binaire. Le soucis est que chaque processeur a ses propriétés propre : il possède un coprocesseur à unité flottantes, ou un accélérateur vectoriel. On cherche alors à rendre le programme *portable* : ne pas le réécrire pour chaque machine. De plus, le langage assembleur est très *bas niveau* : on voudrait pourvoir exprimer des boucles, des conditions, etc, de telle manière à le rendre compréhensible par des humains.

Des *interpréteurs* sont des programmes dont le rôle est de décoder mot-à-mot le langage et le transcrire à la volée en instructions machines. Le soucis est que ces interpréteurs ne sont pas très optimisés : on n'obtient pas le programme le plus rapide pour une machine donnée.

Il existe également des *machine virtuelles*, qui interprètent un assembleur relativement simple sur une machine hôte, comme *Java* et *.net*. Cela permet des optimisations à la compilation et à la volée lors de l'exécution. Il reste cependant toujours une couche intermédiaire.

Enfin, les compilateurs prennent un langage et le traduisent vers un code assembleur directement compréhensible par la machine. La *sémantique* doit être cependant conservée : sur toute entrée du programme, les sorties doivent être identiques. Le compilateur est donc un outils du chercheur HPC permettant de l'aider à déployer son code.

Par contre, un compilateur est moins flexible et dynamique ; certaines optimisations ne pouvant pas être effectuées à la compilation (celle dépendant des valeurs à l'exécution).

1.1 Anatomie d'un compilateur

Le but de ce cours et de développer un compilateur du langage *Tiger* en se basant sur LLVM.

Il existe une multitude de langage sources et d'architecture cibles ; c'est pourquoi on a introduit l'idée de *Représentation intermédiaire* ; qui se divisent en *front-end* (langage vers IR) et *back-end* (IR vers cible). On écrit alors des optimisations qui fonctionneront directement sur le code IR. De plus, le compilateur est plus modulaire, ce qui est un grand avantage. Certains compilateurs utilisent même plusieurs représentation intermédiaires afin d'améliorer la simplicité des optimisations et la maintenabilité du code (encore plus de modularité !).

Le *front-end* se décompose en une analyse *syntaxique*, elle même décomposée en un *lexer* et un *parser*, et une analyse *sémantique*. Le lexer va casser le programmes en tokens (lexèmes) lors que l'analyse syntaxique, qui seront par la suite arrangées dans un arbre de syntaxe abstrait (AST).

L'analyse sémantique se décompose en une phase de *binding* (lier les symboles à leur déclaration), une phase *EscapeChecker* (trouver les symboles d'une fonction définis dans une fonction parent par exemple dans le cas de fonctions imbriqués) puis un *TypeChecker* (vérification des types).

Le *middle-end* va traduire le langage vers le langage intermédiaire, qui permettra ensuite d'appliquer toutes les optimisations LLVM via `opt`.

1.2 Le langage Tiger

Il a été intégré dans le livre *Modern Compiler Implementation* par A. Appel. Il est impératif (ouf), typé selon deux types primitifs : *entier/integers* et *chaines de caractères/strings* (ainsi que *void*).

`let .. in ... end` sépare les déclarations des calculs. Les déclarations se font sous la forme suivante : explicite : `var a : int := 0`, ou implicite : `var b := 1` ou `var c := "hello"`. Des fonctions peuvent également être déclarées : `function get_temperature () : int = thermostat`. Les commentaires se font via les une syntaxe similaire au C : `/* commentaire */`.

Tiger gère également les récursions mutuelles (ce qui est embêtant pour les analyses du binder...). Attention, les fonctions mutuellement récursives ne peuvent être appelés que dans des blocs contigus de déclaration de fonctions (on ne peut pas déclarer de variables entre deux fonction s'appelant l'une l'autre).

Tiger comprend des primitives standards :

— print(s : string)

— print_int(i : int)

— ord(s : string) : int (valeur ASCII)

— chr(i : int) : string (inverse du précédent)

— size(s : string) : int

— et d'autres (concat, substring, not, exit, print_err) (liste explicite).

2 Arbre de syntaxe abstraite

2.1 Définition

Le rôle de l'AST est de donner une représentation que le programme peut comprendre. Les symboles et les mots peuvent avoir des sens différents en fonction de la phase de compilation. L'arbre ne doit pas stocker des informations superflues telles les espaces, les retours à la ligne et les commentaires.

2.2 Parcourir l'arbre

On va utiliser un visiteur. Un arbre est composé de noeuds de différents type : constant ou opération binaire. Un visiteur est un moyen d'évaluer un arbre en utilisant à chaque fois la fonction adaptée au type de l'opérande suivante.

On va utiliser deux méthodes : une `accept` côté arbre et une `visit` côté visiteur. Cela permet de pouvoir faire deux *double dispatch* : appeler une fonction différente en fonction du couple (type de noeud, type de visiteur).

2.3 Le lexer

Le but ici est de différencier les tokens. On va utiliser des expressions régulières. On utilisera *flex* (et par la suite *bison*) pour générer un automate, et même des sous-automates de règles (pour gérer les commentaires ou les chaînes de caractère par exemple).