SE202 : déroulement du cours et bases de compilation

Samuel Tardieu
Année scolaire 2017/2018

Organisation du cours

Objectifs

À la fin de ce module, les étudiants :

- comprendront le processus de compilation et ses différentes étapes ;
- pourront écrire les différentes étapes d'un compilateur basique ;
- sauront s'interfacer avec LLVM ;
- comprendront différents langages assembleur (ARM et Intel).

Déroulement du cours

- Équipe pédagogique : Tarik Graba, Samuel Tardieu.
- La présence est obligatoire aux 8 (7 cette année) séances.
- Chaque absence non justifiée retire 1 point de la note finale.
- Si une séance commence par un QCM, la note de 0 est attribué à une personne absente.
- La note finale est composée d'un contrôle de connaissances sur 7 points, d'un projet sur 9 points et des notes de QCM sur 4 points.

Le projet

- Compilation d'un langage de programme (Tiger) en code assembleur (ARM et Intel).
- C'est un **gros** projet, il faudra travailler en dehors des heures programmées (3ème créneau et travail personnel).
- Le projet sera programmé en C++.
- Certaines phases et structures de données seront fournies.
- Obligation de passer les tests.
- GIT sera utilisé pour toutes les remises:
  - chaque étudiant disposera d'un dépôt personnel, accessible par l'équipe pédagogique ;
  - un dépôt commun sera disponible pour récupérer les sources partagées.

Introduction à la compilation

Le fonctionnement d'un microprocesseur

Un microprocesseur :

- travaille avec des *registres*, en nombre limité, permettant chacun de stocker une valeur ;
- ne comprend que des instructions basiques (charger un registre depuis une adresse mémoire, ajouter deux registres et mettre le résultat dans un troisième, stocker la valeur d'un registre en mémoire) ;
- sait faire des comparaisons entre registres et entre registres et constantes ;
- sait sauter de manière inconditionnelle ou conditionnelle à une autre instruction ;
- peut parfois opérer sur des valeurs en mémoire directement.

Le fonctionnement d'un microprocesseur

Un microprocesseur ne sait pas faire

```
int c = a + 3 * b - 2
```

Ce qu'il sait faire ressemble plus à :

```
r0 <- MEM[a]
r1 <- MEM[b]
r2 <- r1 * 3
r3 <- r0 + r2
r4 <- r3 - 2
MEM[c] <- r4
```

Le fonctionnement d'un microprocesseur

Pour compliquer un peu la chose :

- Chaque concepteur de microprocesseur adopte des instructions différentes (ARM, Intel, Microchip, etc.).
- Au sein des produits d'une même famille, les jeux d'instructions disponibles diffèrent (opérateurs flottants, instructions étendues, etc.).

Les jeux d'instructions peuvent être très différents :

- CISC : peu de registres, instructions complexes ;
- RISC : beaucoup de registres, instructions simples.

L'assembleur à la rescousse

- Il n'est pas nécessaire de saisir les instructions du microprocesseur en binaire, le langage d'assemblage (ou assembleur) permet d'utiliser des mnémoniques (`add`, `mov`, `jmp`) pour les représenter.
- Mais cela reste de très bas niveau : comment représenter des classes, des types, des structures de données complexes ?

Qu'est-ce que la compilation ?

Définition approximative

La compilation est la transformation d'un programme source écrit en langage de haut niveau en instructions compréhensibles par l'ordinateur.

Exemples

- `gcc` compile (entre autres) du code C vers des instructions pour (entre autres) processeur Intel.

Qu'est-ce que la compilation ?

Définition approximative

La compilation est la transformation d'un programme source écrit en langage de haut niveau en instructions compréhensibles par l'ordinateur.

Exemples

- `gcc` compile (entre autres) du code C vers des instructions pour (entre autres) processeur Intel.
- votre projet compilera (si tout se passe bien) du code Tiger vers des instructions pour processeur ARM.

Qu'est-ce que la compilation ?

Définition approximative

La compilation est la transformation d'un programme source écrit en langage de haut niveau en instructions compréhensibles par l'ordinateur.

Exemples

- `gcc` compile (entre autres) du code C vers des instructions pour (entre autres) processeur Intel.
- votre projet compilera (si tout se passe bien) du code Tiger vers des instructions pour processeur ARM.
- `javac` compile du code Java vers... quoi ?

La machine virtuelle Java

- Plutôt que de développer des compilateurs vers toutes les architectures cible, Oracle (anciennement Sun Microsystems) a préféré compiler du code vers une machine *virtuelle*, dont il a défini les instructions de base.

La machine virtuelle Java

- Plutôt que de développer des compilateurs vers toutes les architectures cible, Oracle (anciennement Sun Microsystems) a préféré compiler du code vers une machine *virtuelle*, dont il a défini les instructions de base.
- Quand une nouvelle architecture sort sur le marché, il *suffit* d'écrire un programme (en C par exemple) qui lit les programmes compilés pour la machine virtuelle Java et qui en exécute les instructions.

La machine virtuelle Java

- Plutôt que de développer des compilateurs vers toutes les architectures cible, Oracle (anciennement Sun Microsystems) a préféré compiler du code vers une machine *virtuelle*, dont il a défini les instructions de base.
- Quand une nouvelle architecture sort sur le marché, il *suffit* d'écrire un programme (en C par exemple) qui lit les programmes compilés pour la machine virtuelle Java et qui en exécute les instructions.
- Un programme Java compilé pour la machine virtuelle Java s'exécutera de la même manière sur les différents systèmes pour lesquels existent une machine virtuelle (Linux, Windows, OS X). *Compile once, execute everywhere*

La machine virtuelle Java

- Plutôt que de développer des compilateurs vers toutes les architectures cible, Oracle (anciennement Sun Microsystems) a préféré compiler du code vers une machine *virtuelle*, dont il a défini les instructions de base.
- Quand une nouvelle architecture sort sur le marché, il *suffit* d'écrire un programme (en C par exemple) qui lit les programmes compilés pour la machine virtuelle Java et qui en exécute les instructions.
- Un programme Java compilé pour la machine virtuelle Java s'exécutera de la même manière sur les différents systèmes pour lesquels existent une machine virtuelle (Linux, Windows, OS X). *Compile once, execute everywhere*
- Exemple de programmes Java : Eclipse, Vuze

Qu'est-ce que la compilation ?

Meilleure définition

La compilation est la transformation d'un programme dans un format d'entrée donné dans un programme dans un format de sortie donné.

Exemples

- On peut compiler du C en instructions (*de base*) compréhensibles par le microprocesseur.
- On peut compiler du C en assembleur.
- On peut compiler de l'assembleur en instructions de base.
- On peut compiler du Java en *bytecode* pour la machine virtuelle Java.
- On peut compiler du bytecode en instructions de base.

Compilation et interprétation

Quand on a par exemple un bytecode Java, destiné à la machine virtuelle, on peut :

- l'interpréter, c'est-à-dire lire chaque instruction et faire ce qu'elle dit de faire (facile mais lent à chaque fois) ;
- le compiler en instructions de base (complexe mais exécution rapide après coup) ;
- faire un mélange des deux, et ne compiler que les portions de code qui s'exécutent un grand nombre de fois (*JIT*, ou compilation *Just-In-Time*, à la volée).

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.
- Ils sont compilés en bytecode Java sur l'ordinateur du développeur.

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.
- Ils sont compilés en bytecode Java sur l'ordinateur du développeur.
- Le bytecode Java est compilé dans un autre bytecode, Dalvik, sur l'ordinateur du développeur.

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.
- Ils sont compilés en bytecode Java sur l'ordinateur du développeur.
- Le bytecode Java est compilé dans un autre bytecode, Dalvik, sur l'ordinateur du développeur.
- Avant Android 2.2, le code Dalvik était interprété sur le téléphone.

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.
- Ils sont compilés en bytecode Java sur l'ordinateur du développeur.
- Le bytecode Java est compilé dans un autre bytecode, Dalvik, sur l'ordinateur du développeur.
- Avant Android 2.2, le code Dalvik était interprété sur le téléphone.
- À partir d'Android 2.2, le code Dalvik était compilé à la volée lors de l'exécution.

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Exemple : Android

- La plupart des programmes pour Android sont écrits en Java.
- Ils sont compilés en bytecode Java sur l'ordinateur du développeur.
- Le bytecode Java est compilé dans un autre bytecode, Dalvik, sur l'ordinateur du développeur.
- Avant Android 2.2, le code Dalvik était interprété sur le téléphone.
- À partir d'Android 2.2, le code Dalvik était compilé à la volée lors de l'exécution.
- Depuis Android 5.0, le code Dalvik est compilé en instructions de base lors de l'installation ou la mise à jour de l'application (ART).

On a donc du code portable, que le téléphone ait un microprocesseur ARM ou un microprocesseur Intel.

Autres types de compilation

Il est possible de compiler un langage de programmation vers un autre :

- Le compilateur GHC peut compiler du langage Haskell en instructions de base pour processeurs Intel ou ARM.
- Il peut également compiler du langage Haskell en langage C, pour faciliter le portage vers une nouvelle architecture pour laquelle un compilateur C existe.

La compilation fait partie d'un tout

On utilise souvent le principe de *compilation séparée* :

- Chaque fichier C est compilé en un `.o` (fichier objet) qui contient les instructions à utiliser et les références des symboles (fonctions, variables) définies ou utilisés.
- Chaque fichier Java est compilé en un `.class` qui contient le bytecode et les références.

Une phase d'*édition de liens* recolle ensuite les différents morceaux afin d'en faire un programme complet prêt à s'exécuter sur le système cible.

Le compilateur peut également générer des informations supplémentaires qui aideront au *débogage*.

La compilation n'est pas unique

- Il existe plusieurs suites d'instructions qui donneront le même résultat.
- Différents compilateurs généreront probablement des instructions différentes.
- Un compilateur pourra générer des instructions différentes selon le niveau d'optimisation.

Optimisation

L'optimisation consiste à :

- transformer du code en un autre code produisant des effets équivalents mais plus efficace ;
- tirer partie des redondances dans le code ;
- tirer partie des dépendances entre variables.

Exemple d'optimisation (en C)

```c
int f() {
  return 1 + 2 * 3;
}
```

```
% clang -S -O2 -o - t.c
```

Exemple d'optimisation (en C)

```c
int f() {
  return 1 + 2 * 3;
}
```

```
% clang -S -O2 -o - t.c
```

```
f:
    movl    $7, %eax    ; eax <- 7
    retq                ; retour du résultat dans eax
```

Le compilateur a pu calculer la valeur de l'expression (assez facile). Qu'en est-il des expressions plus complexes avec des boucles ?

Exemple d'optimisation (en C)

```c
int f(int a) {
  int r = 12;
  for (int i = 0; i < 100; i++)
    r += 3 * (a - 1) * i + 7;
  return r - 1;
}
```

```
% clang -S -O2 -o - t.c
```

Exemple d'optimisation (en C)

```c
int f(int a) {
  int r = 12;
  for (int i = 0; i < 100; i++)
    r += 3 * (a - 1) * i + 7;
  return r - 1;
}
```

```
% clang -S -O2 -o - t.c
```

```
f:
  imull $14850, %edi, %eax  ; eax = 14850 * a
  addl  $-14139, %eax       ; eax = 14850 * a - 14139
  retq                      ; retour du résultat dans eax
```

Comment a fait le compilateur ?

```
int f(int a) {
  int r = 12;
  for (int i = 0; i < 100; i++)
    r += 3 * (a - 1) * i + 7;
  return r - 1;
}
```

a été transformé en

```
int f(int a) {
  int r = 12;
  for (int i = 0; i < 100; i++)
    r += a * (3 * i) - (3 * i) + 7;
  return r - 1;
}
```

Comment a fait le compilateur ?

Puis

```
int f(int a) {
  int r = 12;
  for (int i = 0; i < 100; i++)
    r += a * (3 * i) - (3 * i) + 7;
  return r - 1;
}
```

a été transformé en (car $\sum_{i=0}^{99} i = 4950$)

```
int f(int a) {
  int r = 12;
  r += a * 3 * 4950 - 3 * 4950 + 7 * 100;
  return r - 1;
}
```

Comment a fait le compilateur ?

Ensuite

```
int f(int a) {
  int r = 12;
  r += a * 3 * 4950 - 3 * 4950 + 7 * 100;
  return r - 1;
}
```

a été transformé en

```
int f(int a) {
  // return 12 + a * 3 * 4950 - 3 * 4950 + 7 * 100 - 1;
  return 14850 * a - 14139
}
```

Quand faire les optimisations ?

Les optimisations pourraient se faire sur le code d'origine, mais cela a peu d'intérêt (obligation de regénérer du code source valide).

Elles interviennent en général plus tard dans le processus de compilation. LLVM a l'avantage de permettre de sélectionner et visualiser très facilement les différentes optimisations.

Processus de compilation

Lien entre source et cible

- Il est possible de faire un compilateur C vers ARM, C vers Intel, Ada vers ARM, Ada vers Intel, Tiger vers ARM, Tiger vers Intel, etc. Pour n langages et k cibles, cela fait $n * k$ compilateurs complets.
- Il est plus avantageux de mettre en place une représentation intermédiaire (*IR*, pour *Intermediate Representation*). On traduit le langage source vers cet *IR*, puis cet *IR* vers les instructions de la cible.
- Cela permet de changer facilement le langage source ou la cible sans avoir à tout refaire.
- Les structures du langage intermédiaire sont plus faciles à manipuler pour y coder les optimisations.

Traduit on directement la source en IR ?

La représentation intermédiaire est agnostique du langage source. Elle ne permet pas de faire facilement des manipulations dépendantes du langage.

Pour cela, on rajoute une étape intermédiaire : l'*AST* (*Abstract Syntax Tree*, ou arbre syntaxique abstrait). Cet arbre, propre au langage source, est une représentation du programme en entrée. On pourrait par exemple reconstituer le programme en partant de l'AST.

Exemple d'AST

Le fragment de code C suivant

```c
if (a > 3)
  b = 10;
else
  b = 20;
```

pourra par exemple correspondre à l'arbre

```
If(Binop(">", Identifier(a), Number(3)),
   Assign(Identifier(b), Number(10)),
   Assign(Identifier(b), Number(20)))
```

La racine est un nœud `IF` avec trois champs, correspondant respectivement à la condition, la partie à exécuter si la condition est vérifiée et celle à exécuter si elle ne l'est pas.

Exemple d'AST

Autre exemple :

```
int f(int a) {
 a = a + 1;
 return a * 3;
}
```

peut correspondre à

```
Fundecl(f, Type(int), [Param(a, Type(int))],
         [Assign(Identifier(a),
            Binop("+", Identifier(a), Number(1))),
          Return(Binop("*", Identifier(a), Number(3)))
         ]
        )
```

Construction de l'AST

L'AST est généralement construit à partir du programme source à partir de deux outils :

- Un *lexer* découpe le programme d'entrée en *tokens* : nombre, mot clé, identifiant, opérateurs, etc. Son rôle est de transformer le programme source en ces tokens sans savoir ce qu'ils signifient.
- Un *parser* prend les tokens venant du lexer en entrée, et vérifient que leur enchaînement correspond à la *grammaire* du langage.

Lexer

Par exemple, le code C suivant

```
int f(int a) {
  return a + 1;
}
```

donné au lexer produira la suite de tokens suivants :

```
IDENTIFIER("int"), IDENTIFIER("f"), LPAREN,
IDENTIFIER("int"), IDENTIFIER("a"), RPAREN,
LBRACKET, KEYWORD("return"),
IDENTIFIER("a"), PLUS, NUMBER(1), SEMICOLON,
RBRACKET
```

Lexer

Le lexer ne comprend vraiment rien au langage lui-même. Par exemple, si on lui demande d'analyser la ligne

```
int a = b(3};
```

il renverra sans broncher

```
IDENTIFIER("int"), IDENTIFIER("a"), EQUAL,
IDENTIFIER("b"), LPAREN, NUMBER(3), RBRACKET,
SEMICOLON
```

sans s'apercevoir qu'une accolade (`RBRACKET`) ferme la parenthèse (`LPAREN`). Son rôle n'est pas de s'assurer que le programme est syntaxiquement correct (ou qu'il *suit la grammaire du langage*) : c'est au parser de faire cela.

Parser, règles et actions

Le parser est créé à partir d'un ensemble de règles comme

```
assignment : IDENTIFIER EQUAL expression SEMICOLON
  { $$ = new Assign(@2, new Identifier(@1, $1), $3); } ;
expression : expression PLUS expression { ... }
           | NUMBER { ... }
           | LPAREN expression RPAREN { ... } ;
```

Cela signifie que :

- Une affectation (*assignment*) est composé d'un identifiant, d'un signe égal, d'une expression et d'un point-virgule.
- Une expression est composée de trois alternatives avec récursion, autorisant `1 + (2 + 3) + 4`.
- `@2` ici représente l'endroit dans le code source du signe égal (deuxième élément), `$3` la valeur de l'expression (troisième élément).

Priorité et associativité

Il est possible d'associer une priorité (*precedence*) ainsi qu'une associativité aux opérateurs :

- L'opérateur `*` est plus prioritaire que l'opérateur `+`, ainsi `1 + 2 * 3` doit être compris comme `1 + (2 * 3)` et `2 * 3 + 1` comme `(2 * 3) + 1`.
- L'opérateur binaire `-` est associatif à gauche : `10 - 1 - 2` doit être compris comme `(10 - 1) - 2` qui vaut 7 et pas comme `10 - (1 - 2)` qui vaudrait 11.
- L'opérateur `<` n'est pas associatif, on ne peut pas écrire `a < b < c`.

Actions

Après l'exécution du parseur, on se retrouve avec un AST construit récursivement :

- Cet AST est une représentation du programme d'origine.
- Il n'a pas encore été analysé *sémantiquement* : on ne sait pas à quoi correspondent les identifiants par exemple, il est possible que des variables auxquelles on fait référence (des `Identifier`) ne correspondent à aucune déclaration.

Analyse sémantique

La phase d'analyse sémantique essaie de donner du sens à l'AST :

- En parcourant l'arbre, en commençant par le nœud racine et en descendant récursivement dans les nœuds enfants, on va recueillir toutes les déclarations de variables (`int a;` par exemple).
- Lorsqu'on croisera un identifiant dans une expression (le `a` de `a + 3` par exemple), on l'enrichira avec un lien vers sa déclaration dans un champ dédié de `Identifier`.

Ainsi, à la fin de l'analyse sémantique, chaque identifiant fera référence à sa déclaration. Lorsqu'on parlera de `a`, on saura de quel `a` on parle même s'il y en a plusieurs à différents endroits du programme.

Analyse sémantique

Voici le type de lien qu'on trouve après l'analyse sémantique par exemple :

```
int a; <--------------+
int b; <---------+    |
                 |    |
int f() { return b; } |
                      |
int g(int b) { return a + b; }
          ^           |
          |           |
          +-----------+
```

Ces liens seront primordiaux pour savoir quelle registre ou adresse mémoire lire ou modifier lorsqu'on générera le code final.

Analyse sémantique

Lors de l'analyse sémantique, on regarde d'autres choses que les variables :

- On enregistre les déclarations de fonction, et on vérifie lorsqu'elles apparaissent dans une expression qu'elles sont bien définies et qu'elles sont appelées avec le bon nombre d'arguments.
- On enregistre les déclarations de type si le langage permet de définir de nouveaux types.

Ambiguïté

Certains langages comme le C sont ambigus et l'arbre peut nécessité d'être modifié lors de l'analyse. Que signifie `b * a` en C ?

Ambiguïté

Certains langages comme le C sont ambigus et l'arbre peut nécessité d'être modifié lors de l'analyse. Que signifie `b * a` en C ?

```
int a;
int b;
typedef int c;

void f() {
  b * a;
  c * a;
}
```

Ça dépend : il est nécessaire de savoir, pour analyser `* a`, de savoir si ce qui précède est un type (déclaration de variable locale de type pointeur sur `b`) ou une variable (multiplication par `c`).

Récapitulatif

- Le lexer découpe le programme source en tokens (mot clé, identifiant, valeur littérale, opérateur, etc.).
- Le parser construit un arbre syntaxique abstrait (AST) basique (on dit aussi *non décoré*) en vérifiant que la succession de tokens a un sens au regard de la grammaire.
- L'analyse sémantique décore l'AST en ajoutant des liens entre les identifiants et les déclarations correspondantes.
- Pour certaines grammaires ambigües, l'analyse peut amener à modifier l'arbre en fonction de la signification des identifiants.
- Il existe d'autres phases d'analyse, comme l'analyse de type, qui vérifie que les expressions ont des types compatibles avec les déclarations.

C'est ce que vous aurez à faire pour le langage Tiger au tout début de votre projet.

Les étapes suivantes

Une fois l'AST décoré, il restera à (au fur et à mesure du projet) :

- transformer l'AST en représentation intermédiaire (IR) évoqué plus tôt, composé d'un ensemble d'instructions plus basiques (il ne sera plus possible d'avoir des boucles par exemple) en utilisant les fonctions de LLVM ;

Les étapes suivantes

Une fois l'AST décoré, il restera à (au fur et à mesure du projet) :

- transformer l'AST en représentation intermédiaire (IR) évoqué plus tôt, composé d'un ensemble d'instructions plus basiques (il ne sera plus possible d'avoir des boucles par exemple) en utilisant les fonctions de LLVM ;
- travailler sur l'IR pour le mettre sous une forme canonique, par exemple pour évaluer les arguments qu'on passera à une fonction avant d'appeler cette fonction puis en faisant l'appel ;

Les étapes suivantes

Une fois l'AST décoré, il restera à (au fur et à mesure du projet) :

- transformer l'AST en représentation intermédiaire (IR) évoqué plus tôt, composé d'un ensemble d'instructions plus basiques (il ne sera plus possible d'avoir des boucles par exemple) en utilisant les fonctions de LLVM ;
- travailler sur l'IR pour le mettre sous une forme canonique, par exemple pour évaluer les arguments qu'on passera à une fonction avant d'appeler cette fonction puis en faisant l'appel ;
- transmettre cet IR à LLVM, afin qu'il génère le code ;

Les étapes suivantes

Une fois l'AST décoré, il restera à (au fur et à mesure du projet) :

- transformer l'AST en représentation intermédiaire (IR) évoqué plus tôt, composé d'un ensemble d'instructions plus basiques (il ne sera plus possible d'avoir des boucles par exemple) en utilisant les fonctions de LLVM ;
- travailler sur l'IR pour le mettre sous une forme canonique, par exemple pour évaluer les arguments qu'on passera à une fonction avant d'appeler cette fonction puis en faisant l'appel ;
- transmettre cet IR à LLVM, afin qu'il génère le code ;
- demander à LLVM d'optimiser l'IR et de générer le code assembleur.

Mais en attendant

Il est temps de parler du projet.