

Introduction

David Delahaye

David.Delahaye@lirmm.fr

Faculté des Sciences

Master M1 2019-2020



Introduction

Qu'est-ce qu'un compilateur ?

Un compilateur traduit efficacement un langage de haut niveau (adapté à l'esprit humain) vers un langage de bas niveau (conçu pour être exécuté efficacement par une machine).

Exemples

- Langages de programmation (C, Java, etc.) ;
- Langages de description de texte (\LaTeX vers Postscript).

Historiquement

- Domaine très mature (environ 60 ans) ;
- Premier compilateur : Fortran I en 1957.

Un exemple

Maximum d'un tableau d'entiers

```
int main () {
    int *arr, max, n, i;
    printf("Enter the size: ");
    scanf("%d", &n);
    arr = malloc(n * sizeof(int));
    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    max = arr[0];
    for (i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    printf("Max = %d\n", max);
    free(arr);
    return 0;
}
```

Un exemple

Compilation avec gcc sous Intel/Debian

```
.file "max.c"
.section .rodata
.LC0:
.string "Enter the size: "
.LC1:
.string "%d"
.LC2:
.string "Enter %d integers:\n"
.LC3:
.string "Max = %d\n"
.text
.globl main
.type main, @function

main:
.LFB2:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movl $.LC0, %edi
movl $0, %eax
call printf
leaq -20(%rbp), %rax
movq %rax, %rsi      [...]
```

Interprétation et compilation

Analyse syntaxique

- Analyses lexicale et grammaticale ;
- Production d'un arbre de syntaxe abstraite ;
- AST (« Abstract Syntax Tree »).

Interprétation

- Exécution directement à partir de l'AST ;
- Aucune transformation de l'AST.

Compilation

- Génération de code pour une machine ;
- Pour une architecture donnée : compilation native (exemple : C) ;
- Pour une machine virtuelle : compilation bytecode (exemple : Java) ;
- Langages avec les deux types de compilation : OCaml.

Compilation bytecode : un exemple (Java)

Maximum d'un tableau d'entiers

```
class Max {  
    public static void main(String [] argv) {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Enter the size: ");  
        int n = sc.nextInt();  
        int [] arr = new int[n];  
        System.out.println("Enter " + n + " integers :");  
        for (int i = 0; i < arr.length; i++)  
            arr[i] = sc.nextInt();  
        int max = arr[0];  
        for (int i = 1; i < arr.length; i++)  
            if (arr[i] > max)  
                max = arr[i];  
        System.out.println("Max = " + max);  
    }  
}
```

Compilation bytecode : un exemple (Java)

Inspection du fichier objet : javap -c -p Max.class

Compiled from "Max.java"

```
class Max {
    Max();
    Code:
        0: aload_0
        1: invokespecial #1 // Method java/lang/Object."<init>":()V
        4: return

    public static void main(java.lang.String []);
    Code:
        0: new           #2 // class java/util/Scanner
        3: dup
        4: getstatic     #3 // Field java/lang/System.in:Ljava/io/InputStream;
        7: invokespecial #4
            // Method java/util/Scanner."<init>":(Ljava/io/InputStream;)V
       10: astore_1
       11: getstatic     #5 // Field java/lang/System.out:Ljava/io/PrintStream;
       14: ldc           #6 // String Enter the size:
       16: invokevirtual #7
            // Method java/io/PrintStream.print:(Ljava/lang/String;)V
       19: aload_1
       20: invokevirtual #8 // Method java/util/Scanner.nextInt:()I
       [...]
```

Compilation à la volée

Principe

- Traduction dynamique, compilation « Just-In-Time » (JIT) ;
- Première étape de compilation bytecode (portabilité) ;
- Deuxième étape de compilation native lors de l'exécution (efficacité) ;
- Combinaison des avantages de la compilation native et bytecode ;
- Nécessite de pouvoir compiler à l'exécution (compilation dynamique) ;
- Compilation depuis le bytecode moins coûteuse que depuis le source.

Exemples

- VisualWorks (Smalltalk), LLVM, machine virtuelle de .NET, machines virtuelles de Java.

Compilations dynamique et statique

Compilation dynamique

- Dépendance de l'exécutable vis-à-vis de bibliothèques dynamiques ;
- Bibliothèques dynamiques chargées une seule fois en mémoire ;
- Exemple :

```
$ gcc -o max max.c
$ ldd max
        linux-vdso.so.1 => (0x00007ffe9b949000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8d526af000)
        /lib64/ld-linux-x86-64.so.2 (0x000055b86d571000)
```

Compilation statique

- Exécutable « standalone » (aucune dépendance) ;
- Exécutable plus « lourd » (bibliothèques intégrées) ;
- Exemple :

```
$ gcc -static -o max max.c
$ ldd max
        n'est pas un exécutable dynamique
```

Compilation

Difficultés

- Choix des structures de données (AST) ;
- Décomposition en étapes intermédiaires (plusieurs langages) ;
- Bonne connaissance du langage cible (efficacité) ;
- Gestion des erreurs.

Propriétés attendues d'un compilateur

- Correction (le programme traduit fait ce qu'on attend) ;
- Efficacité (du programme traduit).

Le cours

Pourquoi suivre ce cours ?

- Vous n'avez pas le choix ☺ ;
- Pour écrire un programme complexe et élégant (le compilateur) dans un langage de haut niveau ;
- Pour comprendre le fossé entre l'intention humaine et des langages de bas niveau (exécuté par le microprocesseur ou non) ;
- Pour découvrir des techniques et algorithmes d'usage général (transformation d'AST, analyse de flot de données, allocation de registres par coloriage de graphes, etc.).

Ce que nous allons faire

- ① Écrire un compilateur natif d'un petit langage source impératif (qui ressemblera à Pascal et à C) vers du MIPS ;
- ② Écrire un compilateur bytecode d'un langage fonctionnel (Lisp) avec la machine virtuelle correspondante.

Notre compilateur natif

Plusieurs étapes

- ➊ Présentation du langage et de sa sémantique (informelle) ;
- ➋ Rappels de MIPS ;
- ➌ Analyse syntaxique avec ANTLR (PP) ;
- ➍ Typage et sélection d'instructions (de PP vers UPP) ;
- ➎ Création du graphe de flot de contrôle (de UPP vers RTL) ;
- ➏ Explicitation de la convention d'appel (de RTL vers ERTL) ;
- ➐ Analyse de durée de vie (sur ERTL) ;
- ➑ Coloriage de graphe et allocation des registres (de ERTL vers LTL) ;
- ➒ Linéarisation du code (de LTL vers LIN)
- ➓ Réalisation des trames de pile (de LIN vers MIPS).

On suivra les étapes précédentes

- Vous écrirez le compilateur (en Java) ;
- Plusieurs rendus progressifs (TP) ;
- Dates à respecter (pénalités sinon) ;
- Rendus à effectuer sur le site du cours (clé : « compil;2019 ») :

<https://moodle.umontpellier.fr/course/view.php?id=5906>

Le langage

Syntaxe abstraite

- Présentation de la syntaxe abstraite du langage ;
- Description sous la forme d'arbres trop lourde ;
- Donc utilisation de la syntaxe concrète pour ce faire !

Catégories syntaxiques

- Présentation à la BNF (variante) ;
- Catégories (répartition plus ou moins arbitraire) :
 - ▶ Entrées de la grammaire : types, constantes, opérateurs unaires et binaires, cibles d'appels, expressions, instructions, déclarations de fonctions/procédures, programmes.

Le langage

Types

$$\begin{array}{lcl} \tau & ::= & \text{integer} \\ & | & \text{boolean} \\ & | & \text{array of } \tau \end{array}$$

Constantes

$$\begin{array}{lcl} k & ::= & n \\ & | & \text{true} \mid \text{false} \end{array}$$

Le langage

Opérateurs unaires

$$\begin{array}{lcl} \text{uop} & ::= & - \\ & | & \text{not} \end{array}$$

Opérateurs binaires

$$\begin{array}{lcl} \text{bop} & ::= & + \mid - \mid \times \mid / \\ & | & \text{and} \mid \text{or} \\ & | & < \mid \leq \mid = \mid \neq \mid \geq \mid > \end{array}$$

Cibles d'appels

$$\begin{array}{lcl} \varphi & ::= & \text{read} \mid \text{write} \\ & | & f \end{array}$$

Le langage

Expressions

$$\begin{aligned} e &::= k \mid x \\ &\mid \text{uop } e \mid e \text{ bop } e \\ &\mid \varphi(e^*) \\ &\mid e[e] \mid \text{new array of } \tau [e] \end{aligned}$$

Instructions

$$\begin{aligned} i &::= x := e \mid e[e] := e \\ &\mid \text{if } e \text{ then } i \text{ else } i \\ &\mid \text{while } e \text{ do } i \\ &\mid \varphi(e^*) \\ &\mid \text{skip} \\ &\mid i; i \end{aligned}$$

Le langage

Définitions de fonctions/procédures

$$d ::=$$
$$\begin{array}{l} f((x : \tau)^*) [: \tau] \\ [\text{var } (x : \tau)^+] \\ i \end{array}$$

Programmes

$$p ::=$$
$$\begin{array}{l} [\text{var } (x : \tau)^+] \\ d^* \\ i \end{array}$$

Quelques remarques techniques

- Une fonction f retourne un résultat avec une variable (implicite) f ;
- L'appel des sous-programmes se fait par valeur (comme en Java) ;
- Les variables ne sont pas systématiquement initialisées, et on a donc besoin de valeurs par défaut pour tous les types de données :
 - ▶ `default(integer) = 0 ;`
 - ▶ `default(boolean) = false ;`
 - ▶ `default(array of τ) = null.`

Le langage

Quelques remarques sur le langage

- Le langage est typé (on peut/doit vérifier le typage) ;
- Le langage est très réduit et peu expressif ;
- Le langage est moins expressif que C mais plus que MIPS ;
- Mais le langage est Turing-complet : tout algorithme peut être exprimé en utilisant ce langage.

Sémantique du langage

- Syntaxe \neq sémantique ;
- Sémantique = comment s'exécute un programme de ce langage ;
- Sémantique informelle = manuels, « tutorials », exemples ;
- Sémantique formelle = règles mathématiques précises (permettent de raisonner sur la sémantique, et en particulier de prouver que le compilateur est correct).

Un exemple de programme

Que fait le programme suivant ?

```
var  $n$  : integer
```

```
 $f(n : \text{integer}) : \text{integer}$ 
```

```
var  $i$  : integer
```

```
if  $n = 0$  then
```

```
     $f := 1$ 
```

```
else
```

```
     $f := 1;$ 
```

```
     $i := 1;$ 
```

```
    while  $i \leq n$  do
```

```
         $f := f \times i;$ 
```

```
         $i := i + 1$ 
```

```
 $n := \text{read}();$ 
```

```
write( $f(n)$ )
```

Écrire les programmes suivants

- Écrire une fonction qui teste si un entier est un carré ;
- Écrire la fonction factorielle de manière récursive ;
- Écrire un programme qui alloue un tableau d'entiers d'une taille demandée à l'utilisateur, et appelle une procédure pour l'initialiser ;
- Écrire une fonction qui teste si tous les éléments d'un tableau d'entiers passé en paramètre sont positifs.