Compilation

1 Introduction

Les programmes que nous développons sont écrits dans un langage de programmation, par exemple OCaml ou C, qui est construit pour faciliter la vie du programmeur. Cependant, un tel langage n'est pas directement compréhensible par la machine, qui ne comprends que des codes écrits en *langage machine*, une représentation binaire déchiffrée d'une manière particulière qui dépend du processeur de la machine. Les langages de programmation se répartissent dans deux familles selon la manière utilisée pour permettre l'exécution d'un programme : les langages *interprétés* et les langages *compilés*.

Les langages interprétés sont des langages pour lesquels on utilise un programme, appelé *interpréteur*, pour lire, comprendre et exécuter le code. Il est donc nécessaire de coder l'interpréteur dans un autre langage de programmation. Un exemple d'un tel langage est Python, dont l'interpréteur est codé en C. Les langages compilés sont des langages pour lesquels on utilise un programme, appelé *compilateur*, pour traduire le code en langage machine. Là aussi, il est nécessaire de coder le compilateur dans un autre langage de programmation. OCaml et C sont tous deux des langages compilés.

Nous avons ici un problème classique, celui de la poule et de l'œuf, puisqu'il est nécessaire de coder un programme afin d'exécuter le code écrit dans un langage de programmation. On parle de problème d'*amorçage* (*bootstrap* en anglais). Les premiers compilateurs ont été écrits en *assembleur*, un langage représentant le jeu d'instructions du processeur et donc aisément converti en langage machine (on parle d'*assemblage*). Ces compilateurs en assembleur ont permis ensuite de compiler des compilateurs écrits dans d'autres langages. Les compilateurs respectifs de C et d'OCaml sont par exemple désormais écrits en C et en OCaml.

2 Fonctionnement général de la compilation

Le principe de la compilation est donc d'effectuer une traduction d'un langage vers un autre. Il est souvent question de traduire en langage machine un programme écrit dans un certain langage de programmation, mais ce n'est pas une obligation. La compilation est l'opération qui permet de traduire un programme écrit dans un langage *source*[1] vers un langage cible, dit langage *objet*.

La production de fichiers objets passe par plusieurs étapes. Il y a tout d'abord une étape d'*analyse lexicale* afin de reconnaître les différents symboles du langage source. Par exemple, on reconnaît les mots-clés tels que `let`, `for`, `in`, etc., et les symboles comme `=`. Ensuite, il y a l'*analyse syntaxique* qui permet d'établir la structure du programme. Lorsque le mot-clé `let` est suivi un peu plus loin d'un symbole `=` et plus loin encore du mot-clé `in`, on reconnaît par exemple la structure d'une déclaration locale. Le résultat de l'analyse syntaxique est une représentation du programme, interne au compilateur. Cette représentation interne se présente généralement sous la forme d'un *arbre de syntaxe abstraite* (*A.S.T.* pour *Abstract Syntax Tree*). Nous aurons l'occasion de parler d'arbres de syntaxe plus tard. Suivent en général plusieurs étapes, souvent accompagnées de changements de représentation, afin d'analyser et d'optimiser le code (pas toujours de manière sûre...) avant la production des fichiers objets.

1. C'est pour cela que l'on parle de fichiers sources pour les fichiers qui contiennent des programmes.

Si l'on veut produire un programme que la machine peut exécuter, i.e. un *fichier exécutable*, il est en général nécessaire de passer par une étape supplémentaire, appelée l'*édition de liens*. En effet, un programme est souvent structuré en de nombreux sous-programmes dépendant les uns des autres afin de permettre la réutilisation du code sans duplication : on parle de *programmation modulaire*. En pratique, les sous-programmes sont répartis dans plusieurs fichiers sources qu'il faut lier au moment de la compilation du programme principal. Cette étape permet donc de lier les fichiers objets de plusieurs *modules*, ou *bibliothèques*, afin de produire le fichier exécutable du programme principal. Les langages de programmation fournissent en général une *bibliothèque standard*, c'est-à-dire un ensemble de modules regroupant du code considéré comme utile à la majorité des programmeurs.

Nous verrons plus tard qu'il est possible d'écrire plusieurs codes différents fournissant les mêmes fonctionnalités (pas toujours avec la même efficacité, d'ailleurs). C'est pour cela que les langages permettent en général de distinguer des *fichiers d'interface*, contenant une description des fonctionnalités fournies par un module, des *fichiers d'implémentation*, contenant le code source d'un programme réalisant concrètement les fonctionnalités attendues. Il est alors possible de compiler le fichier objet d'un programme dépendant d'une bibliothèque pour certaines fonctionnalités uniquement à l'aide des fichiers d'interface de la bibliothèque, mais il sera nécessaire d'avoir au moment de l'édition de liens le fichier objet correspondant à une implémentation de la bibliothèque afin de produire le fichier exécutable. C'est ce que l'on appelle la *compilation séparée*.

3 Compilation en C et en OCaml

Nous nous intéressons maintenant au fonctionnement de la compilation dans le cas particulier des deux langages au programme.

3.1 Compilation en C

Il existe plusieurs compilateurs pour le langage C. Celui que nous utilisons en cours s'appelle `gcc` (initialement pour *GNU C Compiler* devenu plus tard *GNU Compiler Collection* lorsqu'on l'a étendu à d'autres langages). Son principal concurrent se nomme `clang` et utilise les outils de `LLVM` (pour *Low Level Virtual Machine*), un ensemble d'outils *bas niveau*[2] particulièrement utiles pour la compilation. Ces deux compilateurs sont réputés pour leur efficacité et l'efficacité du code qu'ils produisent, mais certaines optimisations qu'ils appliquent ne sont pas toujours sûres. Il existe un autre compilateur pour le langage C, codé en OCaml et produisant du code un peu moins efficace mais garanti. Ce compilateur est appelé `compcert` pour *compilateur certifié*, c'est-à-dire qu'il a été formellement vérifié à l'aide de l'ordinateur[3] que le programme source en C et le programme assembleur produit par le compilateur calculent bien le même résultat.

2. Par opposition à *haut niveau* : plus les fonctionnalités sont proches du fonctionnement matériel de la machine, plus on est bas niveau. OCaml est donc un langage de plus haut niveau que C par exemple.

3. La démonstration de la correction du compilateur a été effectuée dans l'assistant de preuve Coq, lui-même codé en OCaml.

3.1.1 Usage basique de `gcc`

Pour compiler un programme C à l'aide de `gcc`, il suffit d'utiliser la commande `gcc` sur le nom du fichier source, par exemple :

```
gcc mon_programme.c
```

Cela produira un fichier exécutable nommé par défaut `a.out`. On rappelle qu'un fichier ne peut être compilé en un exécutable que s'il contient une fonction `main`, qui représente le point d'entrée du programme.

Pour donner un autre nom au fichier obtenu, il faut utiliser une option du compilateur. `gcc` vient avec un certain nombre d'options permettant de choisir le nom des fichiers produits, de contrôler les optimisations effectuées sur le programme, de gérer l'affichage des avertissements, etc.... Je vous impose d'utiliser l'option `-Wall` pour afficher tous les avertissements possibles, et vous pouvez choisir le nom de l'exécutable produit à l'aide de l'option `-o` suivie du nom désiré. Par exemple, on pourrait écrire

```
gcc -Wall -o mon_programme.exe mon_programme.c
```

3.1.2 Compilation séparée

Les fichiers d'interface en C sont les fichiers d'en-tête dont l'extension est `.h`. Pour indiquer au compilateur qu'une bibliothèque est nécessaire à la compilation de votre fichier source, il faut inclure dans le fichier une directive déclarant cette bibliothèque. Une telle directive s'écrit `#include <nom.h>` pour les biliothèques de la bibliothèque standard de C et `#include "nom.h"` pour les bibliothèques que vous produisez vous-mêmes. Ces directives sont traitées par un *préprocesseur* qui inclut directement le contenu du fichier d'en-tête dans le code source avant la compilation.

Un fichier d'en-tête `nom.h` contient les déclarations des types [4], des constantes et des fonctions qui sont implémentées dans le fichier source `nom.c`. Pour déclarer une fonction sans donner son implémentation (on parle de déclaration de la signature de la fonction), on remplace simplement le bloc de code du corps de la fonction par un point-virgule. Par exemple,

```
int max_2_int(int x, int y);
```

est la déclaration de la signature d'une fonction `max_2_int` censée calculer le maximum de deux entiers. Comme un fichier d'en-tête peut être inclus dans plusieurs fichiers sources utiles à la compilation du programme principal, on utilise des directives spéciales pour éviter les déclarations multiples. Le fichier `nom.h` s'écrit alors :

```
#ifndef NOM_H
#define NOM_H
  // code de nom.h
#endif
```

La première directive permet de vérifier que la constante `NOM_H` n'est pas définie (donc que le fichier n'a pas encore été inclus). Si c'est bien le cas, alors la constante est définie grâce à la deuxième directive, ce qui évitera d'exécuter le code de ce fichier lors de la prochaine inclusion. La troisième directive sert seulement de délimiteur au code qui n'est exécuté que si la constante n'est pas définie.

4. Nous verrons plus tard les déclarations de types

Lorsque l'on dispose d'un fichier source `fichier.c` et des fichers d'en-tête qui lui sont nécessaires, on peut alors produire le fichier objet correspondant à l'aide de la commande

```
gcc -c fichier.c
```

qui produit le fichier objet `fichier.o`. Lorsque les fichiers objets de tous les fichiers sources sont obtenus, on peut alors les lier pour produire le fichier exécutable à l'aide de la commande

```
gcc -o mon_programme.exe fichier_objet_1.o ... fichier_objet_n.o
```

3.2 Compilation en OCaml

Le langage OCaml est distribué avec deux compilateurs. Le compilateur nommé `ocamlc` produit du *bytecode* (code en octet), c'est-à-dire un format de code binaire qui n'est pas directement compréhensible par la machine mais qui devra être exécuté par une machine virtuelle (grâce à la commande `ocamlrun` dans le cas d'OCaml). Le compilateur nommé `ocamlopt` produit un fichier directement exécutable. Comme en C, un fichier ne peut être compilé en un exécutable que s'il contient une déclaration d'un objet de type `unit` qui désigne une instruction à exécuter lorsque le programme est lancé :

```
let main =
   (* expression de type unit *)
```

Il n'est par ailleurs pas nécessaire de nommer cet objet. Le code suivant est équivalent :

```
let _ =
   (* expression de type unit *)
```

En fait, il est même possible d'en déclarer plusieurs : les instructions seront exécutées dans l'ordre de leur déclaration.

Les deux compilateurs d'OCaml ont, comme `gcc`, une option `-o` permettant de définir le nom du fichier produit et une option `-c` permettant de produire un fichier objet sans faire la génération de lien. La compilation sans l'option `-c` produit un fichier nommé `a.out` par défaut. Les fichiers objets produits par `ocamlc` ont pour extension `.cmo` et ceux produits par `ocamlopt` ont pour extension `.cmx`. La démarche pour la compilation séparée avec ces deux compilateurs est la même que celle pour `gcc`. La différence principale est la gestion des modules et des fichiers d'interface.

Chaque fichier `nom.ml` est considéré comme un module contenant toutes les déclarations globales du fichier. On peut utiliser sans déclaration supplémentaire les fonctions d'un module dans un autre fichier : si le fichier `mon_module.ml` déclare une fonction `ma_fonction`, on peut l'utiliser dans un autre code source en écrivant `Mon_module.ma_fonction` (c'est le principe de l'usage des fonctions sur les listes et les tableaux que nous avons vues jusqu'ici). Il est possible d'utiliser cette fonction sans avoir à préfixer son nom par le nom `Mon_module` (donc en écrivant simplement `ma_fonction`) grâce à la directive `open Mon_module` en début de fichier.

Un fichier d'interface en OCaml a pour extension `.mli`. Il n'est pas nécessaire de le protéger contre les inclusions multiples par des directives comme pour les fichiers d'en-tête de C. Les fichiers d'interface sont facultatifs en OCaml : nous avons vu que nous pouvons utiliser une fonction d'un module sans directive particulière. Écrire un fichier d'interface `mon_module.mli` permet en fait de restreindre l'ensemble des déclarations accessibles depuis l'extérieur du module. Par exemple, si la fonction `ma_fonction` utilise une fonction utilitaire déclarée globalement `ma_fonction_aux` mais que cette dernière n'est pas déclarée dans

le fichier d'interface, alors il sera impossible de s'en servir en dehors du module, même en écrivant `Mon_module.ma_fonction_aux`. Un fichier d'interface ne contient que les déclarations de types (que nous verrons plus tard), des constantes et des fonctions que l'on souhaite rendre accessibles. Pour déclarer les signatures des objets que l'on souhaite rendre accessibles, on utilise la syntaxe :

```
val nom_objet : type_objet
```

Par exemple, on peut écrire un fichier `mon_module.mli` contenant la signature suivante :

```
val ma_fonction : int -> int -> int
```

si `ma_fonction` est une fonction prenant deux entiers en paramètres et renvoyant un entier.