Prérequis et sources

Ce cours suppose que vous ayez compris les cours

- algorithmique et programmation,
- programmation fonctionnelle en Objective Caml.

Ce cours s'inspire de ceux de

- Martin Odersky (École Polytechnique Fédérale de Lausanne),
- Luc Maranget, Didier Rémy (INRIA, École Polytechnique),
- François Pottier, Xavier Leroy et Michel Mauny (INRIA).

Pourquoi étudier les compilateurs ?

Très peu de professionnels écrivent des compilateurs.

Alors pourquoi apprendre à construire des compilateurs ?

- Un bon informaticien comprend les langages de haut niveau ainsi que le matériel.
- Un compilateur relie ces deux aspects.
- C'est pourquoi comprendre les techniques de compilation c'est comprendre l'interaction entre les langages de programmation et les ordinateurs.
- Beaucoup d'applications contiennent de petits langages pour leur configuration ou rendre souple leur contrôle (macros Word, scripts pour le graphisme et l'animation, les descriptions des structures de données etc.)

Pourquoi étudier les compilateurs ? (suite)

- Les techniques de compilation sont nécessaires pour l'implantation de tels langages.
- Les formats de données sont aussi des langages formels (langages de spécification de données), tels que HTML, XML, ASN.1 etc.
- Les techniques de compilation sont nécessaires pour lire, traiter et écrire des données, mais aussi pour migrer des applications (réingénierie).
- À part cela, les compilateurs sont d'excellents exemples de grands systèmes complexes
  - qui peuvent être spécifiés rigoureusement,
  - qui ne peuvent être réalisés qu'en combinant théorie et pratique.

Le rôle d'un compilateur

- Le rôle d'un compilateur est de traduire des textes d'un langage *source* en un langage *cible*.
- Souvent le langage source est plus abstrait (p.ex. langage de programmation) que le langage cible (p.ex. assembleur).
- Néanmoins, on nomme parfois les compilateurs des *traducteurs* lorsqu'ils traduisent des programmes entre langages de même niveau d'abstraction.
- Une partie du travail d'un compilateur est de vérifier la validité du programme source.
- La spécification d'un compilateur est constituée par
  - une spécification des langages source et cible,
  - une spécification de la traduction des programmes de l'un vers l'autre.

Langages

- Formellement, un langage est un ensemble de *phrases*. Une phrase est une suite de *mots*. Un mot est une suite de *caractères* appartenant à un *alphabet* (ensemble de symboles fini non vide).
- Chaque phrase possède une structure qui peut être décrite par un arbre.
- Les règles de construction d'une phrase s'expriment à l'aide d'une *grammaire*.

Ainsi,

- les phrases d'un langage de programmation sont des programmes ;
- les mots d'un programmes sont appelés *lexèmes* ;
- les lexèmes suivent aussi des règles qui peuvent être données par une grammaire.

Structure simplifiée d'un compilateur

1. Analyse lexicale : texte source $\mapsto$ suite de lexèmes ;
2. Analyse syntaxique : suite de lexèmes $\mapsto$ arbre de syntaxe abstraite ;
3. Analyses sémantiques sur l'arbre de syntaxe abstraite :
   - 3.1 Vérification de la portée des identificateurs (gestion des environnements) ;
   - 3.2 Vérification ou inférence des types (optionel) : arbre $\mapsto$ arbre décoré.
4. Production de code intermédiaire : arbre [décoré ?] $\mapsto$ code intermédiaire ;
5. Optimisations intrinsèques du code intermédiaire ;
6. Production de code cible (objet), p.ex. assembleur ;
7. Optimisations du code cible (dépendantes de la machine cible) ;
8. Édition de liens (statique) : code cible + bibliothèques $\mapsto$ code exécutable.

Remarques

- L'analyseur lexical (*lexer*) reconnaît les espaces, les caractères de contrôle et les commentaires mais n'en fait pas des lexèmes (*tokens*).
- L'arbre de syntaxe abstraite est appelé *Abstract Syntax Tree* (AST).
- Les trois premières étapes constituent la *phase d'analyse*. Les restantes constituent la *phase de synthèse* (de code). On parle aussi de *phase frontale* pour les étapes jusqu'à la production de code intermédiaire incluse, et de *phase finale* pour les suivantes.
- L'association d'un type aux constructions du langage s'appelle le *typage*. Il garanti que les programmes ne provoqueront pas d'erreurs à l'exécution pour cause d'incohérence sur leurs données (néanmoins, une division par zéro restera possible). Selon la finesse du typage, plus ou moins de programmes valides sont rejetés.

Interprétation

Un *interprète* transforme un fichier source en une donnée, par exemple un arbre de syntaxe abstraite ou du code intermédiaire, que l'on passe ensuite à un programme, dit *machine virtuelle*, qui l'exécutera en mimant (donc abstraitement) une machine réelle (physique).

Remarques

- Non à l'horrible anglicisme « interpréteur » !
- Le code intermédiaire est parfois appelé *byte-code*.
- L'arbre de syntaxe abstraite n'est pas toujours construit, ou pas complètement, selon les langages ou les stratégies d'implantation.
- Un interprète contient donc les premières phases d'un compilateur.

Lexique et syntaxe

- L'ensemble des lexèmes d'un langage est appelé *lexique*.
- La *syntaxe concrète* décrit comment assembler les lexèmes en phrases pour constituer des programmes. En particulier,
  - elle ne donne pas de sens aux phrases ;
  - plusieurs notations sont possibles pour signifier la même chose, p.ex. en OCaml : 'a' et '\097', ou (...) et **begin** ... **end**.
  - ce qu'elle décrit est **linéaire** (le code source est du texte) et utilise généralement des parenthèses.
- La *syntaxe abstraite* décrit des **arbres** qui capturent la structure des programmes (p.ex. les imbrications correspondent à des sous-arbres).

Étude d'une calculette

- **Syntaxe concrète** (dans le style *Backus-Naur Form* (BNF))

```
Expression ::= integer
             | Expression BinOp Expression
             | "(" Expression ")"
BinOp ::= "+" | "-" | "*" | "/"
```

  Pour l'analyse syntaxique, les priorités des opérateurs est celle habituelle.

- **Syntaxe abstraite** (en OCaml)

```
type expr = Const of int
          | BinOp of bin_op * expr * expr
and bin_op = Add | Sub | Mult | Div;;
```

Remarque On utilisera parfois la police des arbres de syntaxe abstraite pour le code source. Par exemple (1+7)*9 (polices mêlées) au lieu de `(1+7)*9`.

Un exemple d'expression arithmétique

- Il faudrait définir l'ensemble de lexèmes dénoté par `integer` dans la grammaire.
- L'analyse lexico-syntaxique transforme l'extrait `"(1+2)*(5/1)"` en syntaxe concrète ou `"(1 +2)*(5 / 1)"` en le *terme* (c.-à-d. la valeur OCaml)

  BinOp (Mult, BinOp (Add, Const 1, Const 2), BinOp (Div, Const 5, Const 1))

  qui est le parcours préfixe gauche de l'arbre de syntaxe abstraite