Concurrence et Compilation

Compilation

Vendredi 8.00 – 10.00

Cours: CO3

Exercices et Projet: IN1

Christine Röckl, LAMP, INR 320

christine.roeckl@epfl.ch

http://lampwww.epfl.ch/teaching/cec/

Cours

Introduction aux fondements de la compilation.

Organisation et manipulation des données. (Util dans d'autres domaines!)

Pour en savoir plus:

- A. Appel. *Modern Compiler Implementation in Java.* Addison-Wesley, 1998.

  Introduction bien illustrée. A la base du cours.

- A. Aho, R. Hopcroft, J. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

  Le grand classique. Très théorique.

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

  Techniques de programmation orienté-objet.

Exercices, Projet

Pour bien comprendre la théorie, le mieux est de s'en occuper en pratique, c'est-à-dire en écrivant son propre compilateur.

Pas obligatoires, pas de "solution" officielle. Mais des questions sont toujours bienvenues. Travail en petits groupes (2 – 3 personnes) conseillé.

Examen:

Par écrit à la fin de sémestre dans le style des exercices.

Basé sur le matériel présenté dans le cours, les exercices, et le projet.

Périodes

1. **Introduction:** survol, phases.
2. **Analyse lexicale:** la micro-syntaxe.
3. **Analyse syntaxique:** la macro-syntaxe.
4. **Grammaires, langages:** théorie, problèmes, techniques.
5. **Analyse des noms:** portées, tables de symboles.
6. **Analyse des types.**
7. **Génération de code.**
8. **Outils:** comment automatiser la création de compilateurs?
9. **JLex, JavaCup:** le côté pratique.
10. **Langages réguliers:** grammaires, expressions régulières, automates finis déterministes (AFD) et non-déterministes (AFN).
11. **Langages non-contextuels:** grammaires, automates à pile.

Les périodes seront accompagnées par des exercices et/ou du travail sur le projet.

Introduction

Qui aimerait implémenter un algorithme pour la multiplication de deux matrices dans un langage d'assemblage? Il est beaucoup plus facile de le faire (et moins probable de faire des erreurs) si on utilise un langage de haut niveau.

$\Rightarrow$ Il est donc nécessaire d'avoir des compilateurs pour traduire les programmes dans des langages machine appropriés.

Mais pourquoi ne pas laisser la problématique à quelques spécialistes?

$\Rightarrow$ Des programmes traitant des données doivent les lire d'abord, ce qui nécessite des techniques de compilation.

$\Rightarrow$ Dans beaucoup de domaines, des petits langages spécialisés doivent être développés et implémentés.

$\Rightarrow$ La compilation nous apprend comment organiser des données, comment ensuite analyser ces données, et comment les manipuler, ainsi que comment les émettre (sous une nouvelle forme).

Survol de cette période:

- Les tâches d'un compilateur.
- Les tâches d'un interprète.
- Survol: les phases de compilation.
- Grammaires: un outil pour décrire des langages (de programmation).

Les tâches d'un compilateur

- Traduction d'un programme écrit dans un **langage source** en un programme en **langage cible (langage objet)**. Souvent le langage source est un langage de haut niveau, et le langage cible un langage de machine.
- Analyse syntaxique et sémantique. Le compilateur vérifie qu'un programme satisfait les restrictions du langage de source. Par exemple, que les variables et fonctions d'un programme sont définies avant l'usage.
- Souvent optimisation de code.

Les tâches d'un interprète

- Transformation d'un programme écrit dans un **langage source** en une **représentation interne** qui est **exécutée** tout de suite. Pas de code cible.

Comparaison

- L'éxecution d'un programme compilé est généralement plus rapide que son interprétation.
- La phase de l'optimisation permet souvent de rendre le programme beaucoup plus efficace.
- Pendant le développement il est plus agréable de pouvoir tester le code écrit tout de suite sans devoir le compiler avant (en plus, une compilation prend du temps).
- La génération de code d'un compilateur doit être adaptée à la machine utilisée. (Portabilité!)

Langages compilés, langages interprétés

Evidemment, compilation et interprétation ont leurs applications. Pour combiner l'efficacité du code compilé avec la portabilité qu'offrent les interprètes, on peut utiliser une combinaison des deux méthodes: les **machines virtuelles**.

Langages compilés:

- C, C++

Langages généralement interprétés

- LISP, SCHEME

Langages qui combinent les deux:

- D'abord compilation, y compris optimisations.
- Ensuite interprétation du code cible sur une **machine virtuelle**.
- Java, Pascal

Les phases de compilation

entrée	phase	sortie
code de source	**analyse lexicale (scanner)**	séquence de lexèmes
séquence de lexèmes	**analyse syntaxique (parser)**	arbre syntaxique
arbre syntaxique	**analyse sémantique**	arbre syntaxique attribué
arbre syntaxique attribué	**génération de code intermédiaire**	séquence de code intermédiaire
séquence de code intermédiaire	**optimisations**	séquence de code intermédiaire
séquence de code intermédiaire	**génération de code cible**	séquence de code cible

Les phases n'ont pas forcément lieu l'une après l'autre, certaines s'effectuent parfois en parallèle.

Les structures de données intermédiaires n'existent parfois jamais dans leur entier.

S'il n'y a pas d' optimisations, il est inutile de générer du code intermédiaire.

L'interprétation peut s'effectuer sur l'arbre syntaxique (nous verrons un exemple dans les exercices) ainsi que sur le code intermédiaire. Pour des langages très simples, il n'est même pas nécessaire de construire un arbre syntaxique: l'interprétation est effectué en même temps que l'analyse syntaxique.

Les phases d'analyse (front-end)

Analyse lexicale (“lit les charactères, forme des mots”)

Lit le code source charactère par charactère.

Livre des lexèmes à l'analyseur syntaxique: mots clés (“`if`” ⇒ `IF`), nombres (“`23`” ⇒ `NUM`), identificateurs (“`x11`” ⇒ `ID`), etc.

Elimine les commentaires et les espaces blancs.

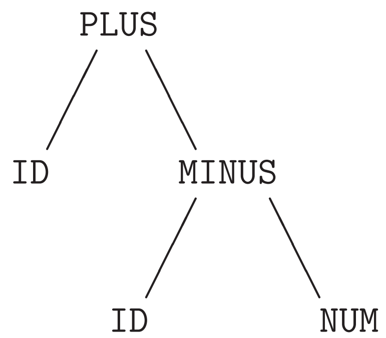
Analyse syntaxique (“lire les mots, former des phrases”)

Construit un arbre syntaxique des lexèmes livrés par l'analyseur lexicale.

Elimine la ponctuation (parenthèses, virgules) qui est implicite dans la structure de l'arbre.

`ID PLUS LPAREN ID MINUS NUM RPAREN`

⇒



Attaché aux feuilles marquées `ID`, il y a l'information sur le nom que porte l'identificateur (“`x11`”).
Aux feuilles marquées `NUM`, on trouve l'information sur les valeurs (“`23`”).

Analyse sémantique (”examine la structure des phrases”)

Examine les noms, les portées, et les types.

Crée et se sert de la **table des symboles**. Celle-ci contient toutes les informations (portée, type, valeur lors d'une interprétation, etc.) sur les symboles.

Langages et grammaires

Un **langage** est un ensemble (fini ou infini) de mots sur un **alphabet**.

$L_1 = \{ a, aa, ab, \epsilon \}$ et $L_2 = \{ a^n b \mid n = 0, 1, \ldots \}$ sont des langages sur l'alphabet $\{ a, b \}$. Le mot ϵ signifie le **mot vide**. Comme pour tous les ensembles, on peut par exemple prendre l'intersection $L_1 \cap L_2 = \{ ab \}$.

Une **grammaire** G = (N, T, S, R) est un moyen pour décrire un langage.

N est l'ensemble des **symboles non-terminaux**. Généralement représentées par des lettres majuscules.

T est l'ensemble des **symboles terminaux** (l'alphabet). Généralement représentées par des lettres minuscules; parfois entre guillemets.

$S \in N$ est le **symbole de départ**.

R est l'ensemble des **règles** $m \to n$ ($m, n \in (N \cup T)^*$) pour produire/accepter (selon le point de vue) des mots. Dans le cours, nous n'allons parler que des grammaires ou $m \in N$. Règles souvent aussi sous la forme m ::= n.

Une dérivation est une chaîne d'applications de règles (**productions**) partant du symbole de départ. Une telle dérivation définit un **arbre (de dérivation)**.

Par exemple, la grammaire ($\{ S \}, \{ a, b \}, S, \{ S \to \epsilon \mid a \mid aa \mid ab \}$) décrit L_1, tandis que ($\{ S, A \}, \{ a, b \}, S, \{ S \to Ab, A \to \epsilon \mid aA \}$ décrit L_2. Mais ($\{ S \}, \{ a, b \}, S, \{ S \to aS \mid b \}$ décrit aussi L_2.

Attention: L'ensemble des symboles non-terminaux d'une grammaire lexicale correspond à l'alphabet de la grammaire syntaxique.

Convention: En parlant des grammaires, nous n'allons souvent donner que les règles. En général, il est clair quels sont les symboles terminaux et non-terminaux.

Un exemple

$$
\begin{array}{rcl}
\text{Stm} & \rightarrow & \text{id ":=" Exp} \mid \text{"print" Exp} \mid \text{Stm ";" Stm} \\
\text{Exp} & \rightarrow & \texttt{num} \mid \texttt{id} \mid \text{Exp Bop Exp} \\
\text{Bop} & \rightarrow & \text{"+"} \mid \text{"-"} \mid \text{"}*\text{"} \mid \text{"/"} \\
\\
\texttt{num} & \rightarrow & \texttt{chiffre} \mid \texttt{chiffre num} \\
\texttt{chiffre} & \rightarrow & \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"} \\
\texttt{id} & \rightarrow & \texttt{lettre} \mid \texttt{id num} \mid \texttt{id lettre} \\
\texttt{lettre} & \rightarrow & \text{"a"} \mid \ldots \mid \text{"z"} \mid \text{"A"} \mid \ldots \mid \text{"Z"}
\end{array}
$$

Une dérivation pour "a := 15 ; print a ; print a":

Stm → Stm ; Stm → id := Exp ; Stm → id := num ; Stm → id := num ; Stm ; Stm → id := num ; print Exp ; Stm → id := num ; print Exp ; print Exp → id := num ; print id ; print Exp → id := num ; print id ; print id → id := chiffre num ; print id ; print id → id := chiffre chiffre ; print id ; print id → id := chiffre chiffre ; print lettre ; print id → id := chiffre chiffre ; print lettre ; print lettre → lettre := chiffre chiffre ; print lettre ; print lettre → a := chiffre chiffre ; print lettre ; print lettre → a := 1 chiffre ; print lettre ; print lettre → a := 15 ; print lettre ; print lettre → a := 15 ; print a ; print lettre → a := 15 ; print a ; print a

La grammaire est **ambiguë**: par exemple, il existe des arbres différents pour "a := 15 ; print a ; print a":