Compilation

Introduction

Principes, Phases de compilation, ...

SI4 — 2018-2019

Erick Gallesio

Introduction

- **Compilation**: traduction d'un langage vers un autre:
  - *Le langage de départ est appelé langage source*
  - *le langage d'arrivée est appeleé langage cible*
- À l'origine, les techniques de compilation on été développées pour traduire
  - *un programme exprimé dans un langage source de haut niveau (e.g. C, C++)*
  - *vers un programme exprimé dans un langage cible de plus bas niveau (e.g. langage d'assemblage)*
- Mais pas que …
  - *le langage source peut ne pas être un langage de programmation:*
    - Analyse de langages de description (XML, JSON, CSS)
    - Analyse de mini langages ad hoc (*init files*)
  - *le langage cible non plus:*
    - enjoliveur de programme

Pourquoi un cours de compilation?

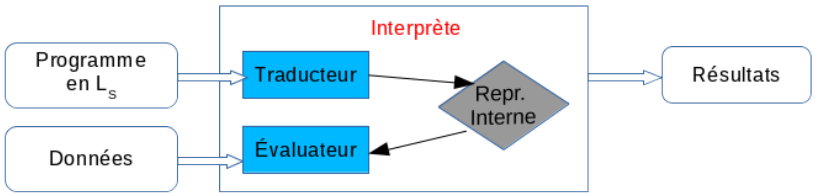
- Permet de mieux comprendre
  - *les principes de fonctionnement des langages de programmation,*
  - *les techniques employée suivant les classes de langages,*
  - *ce qui se passe "behind the scene",*
  - *à quoi servent les notions "theoriques" vues dans d'autres cours (automates, expressions régulières, …)*
- Utilisation d'outils de construction d'analyseurs utiles dans de nombreux contextes:
  - *lex/flex (pour l'analyse lexicale)*
  - *yacc/bison (pour l'analyse syntaxique*
- Implémentation de petits DSLs (*Domain Specific Language*) est souvent nécessaire

Bref, un ingénieur sera inévitablement confronté des problèmes de compilation (même si il n'écrira probablement pas de compilateur classique de bout en bout).

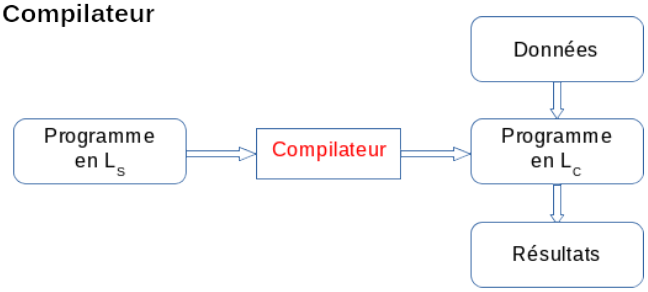
Compilateur vs Interprète (1/2)

Ce sont deux modes que l'on rencontre souvent pour implémenter des langages de programmation.

Interprète



Compilateur



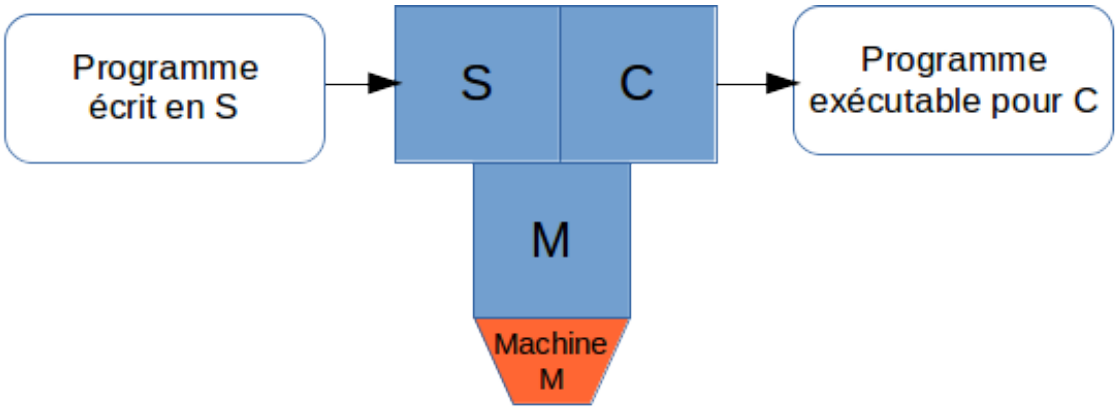
Compilateur vs Interprète (2/2)

- Compilateur
  - *production du code machine ⇒* ***efficace***
  - *comme la traduction n'est faite qu'une fois, possibilité d'avoir des optimisations coûteuses (en temps de calcul)*
  - *dépendant de la machine cible ⇒* ***peu portable****.*
- Interprète
  - ***moins efficace***
    - traduction à chaque exécution
    - évaluation plutôt qu'exécution directe (JIT pondère ce point)
  - *programme peu dépendant de la machine cible ==> *portable***
  - *possibilité de manipuler le code source (****méta programmation****)*
  - *possibilité de travailler sur la représentation interne (****introspection****)*

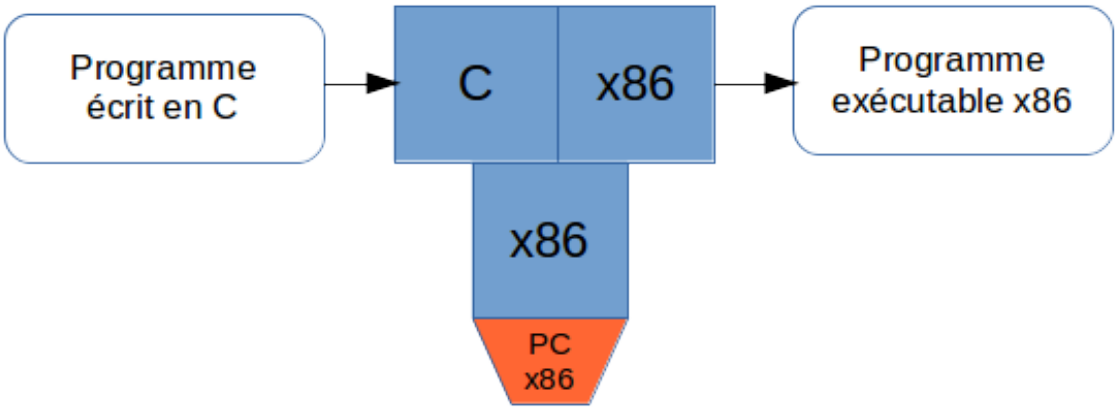
Généralement, un programme interprété est entre 10 et 100 fois moins efficace qu'un programme équivalent compilé.

Principe de compilation

Un compilateur qui traduit un langage **S** vers un langage **C** est un programme écrit pour une machine **M** est représenté par le schéma.



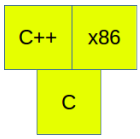
Exemple :



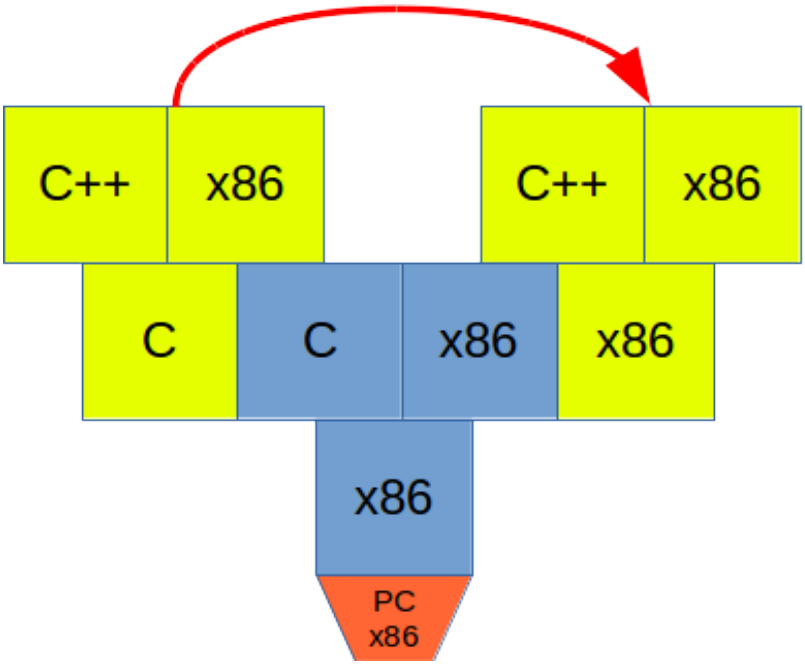
Un compilateur C pour PC écrit en assembleur x86

Obtention d'un compilateur

Supposons que l'on a écrit (en C) un compilateur C++ qui produit du x86.



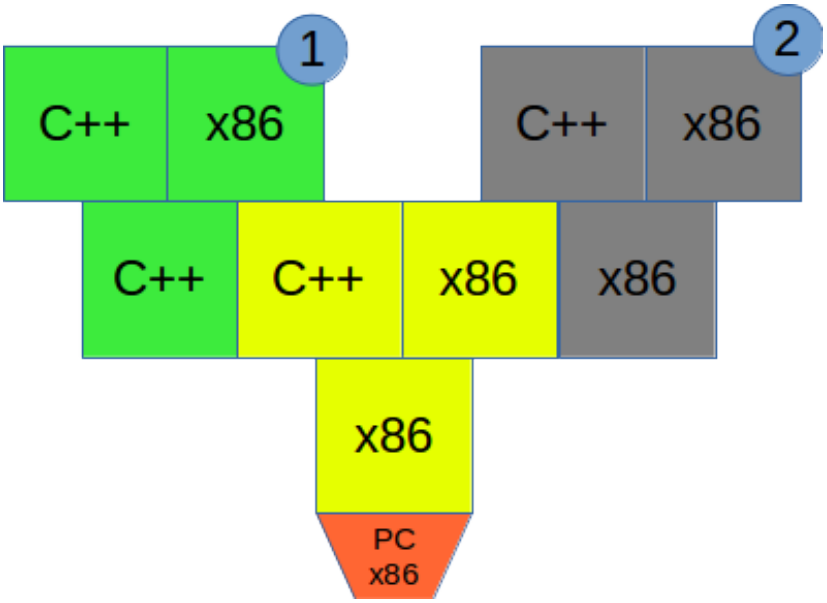
Si on compile ce nouveau compilateur avec le précédent:



On obtient un compilateur C++ qui tourne sur x86

Principe de bootstrap (1/4)

On peut modifier notre nouveau compilateur en le réécrivant en C++. On obtient un nouveau compilateur qui peut être compilé avec le compilateur précédent.



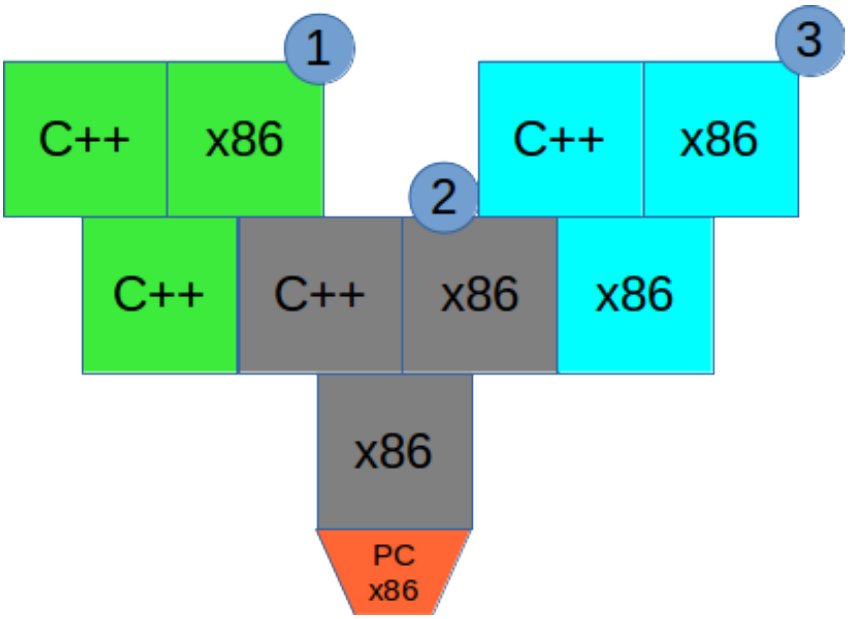
On a donc 2 compilateurs C++ qui produisent du x86:

- Le compilateur **1** écrit en C++
- le compilateur **2** écrit en x86

On peut donc maintenant recompiler le compilateur **1** avec le compilateur **2**.

Principe de bootstrap (2/4)

On obtient:

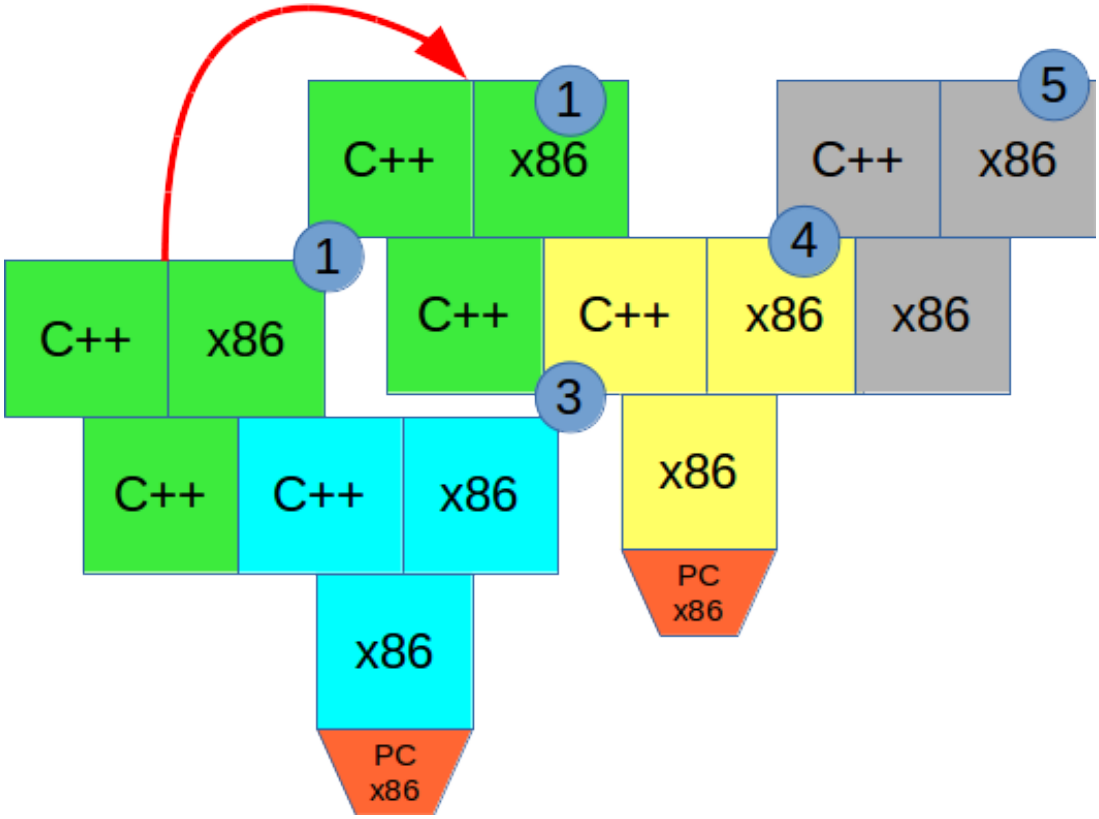


où le compilateur **3**:

- est un compilateur C++ produisant du x86 obtenu à partir d'un compilateur écrit en C++
- est plus facilement modifiable que le compilateur initial qui étéait écrit en C
- permet de tester le compilateur sur un programme conséquent (un compilateur C++!!).

Principe de bootstrap (3/4)

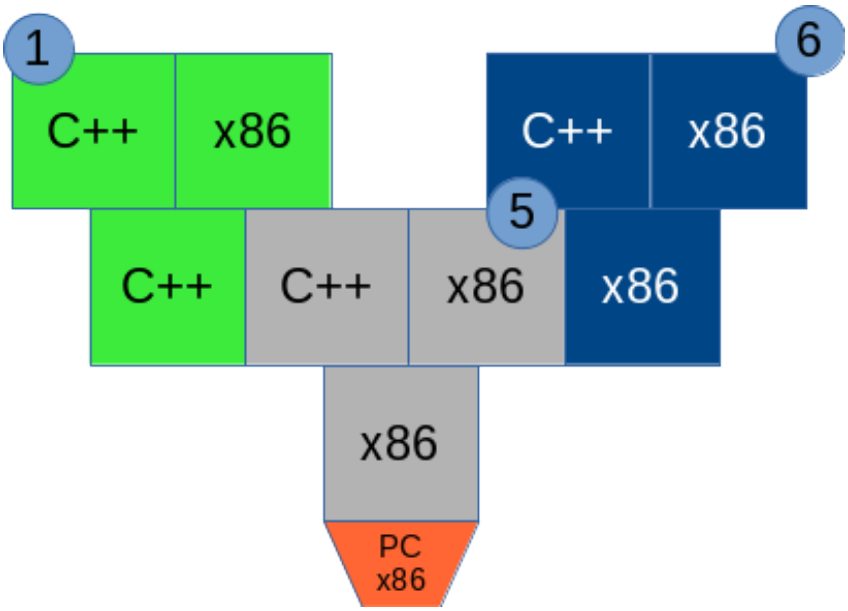
Pour vérifier que notre compilateur est complet et correct, nous pouvons maintenant enchaîner les compilations pour obtenir:



- **4** est obtenu en compilant **I** notre compilateur optimisé avec **3**
- **5** est le résultat de la compilation de **I** avec un compilateur issu de **I**, mais qui a été compilé par le compilateur original (en x86)

Principe du bootstrap (4/4)

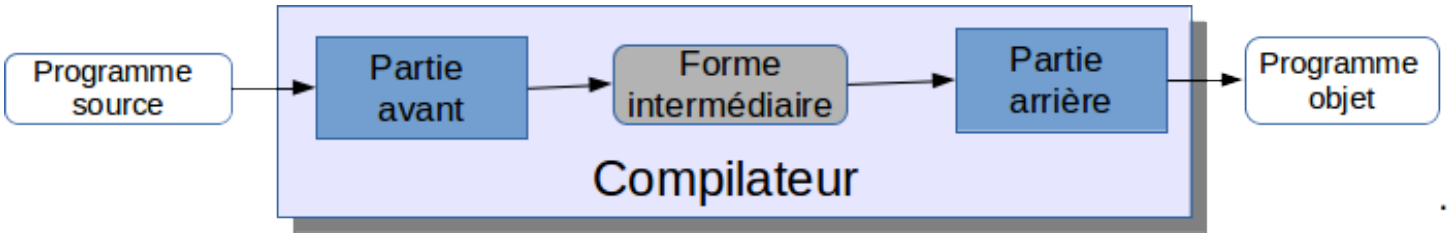
En recompilant une dernière fois **1** avec **5**, on obtient:



- **6** est un compilateur issu de **1** dont la compilation n'a été réalisée que par un compilateur issu de **1**
- Si les binaires de **5** est **6** sont identiques bit à bit, notre compilateur est **correct**.

Compilateur modulaire (1/2)

- Le code produit par un compilateur dépend de la machine cible.
- Pour faciliter le portage du compilateur vers d'autres architectures, on passe souvent par une **forme intermédiaire**.



Partie avant:

- toutes les analyses:
  - *lexicale,*
  - *syntaxique,*
  - *sémantique,*
  - …

Partie arrière:

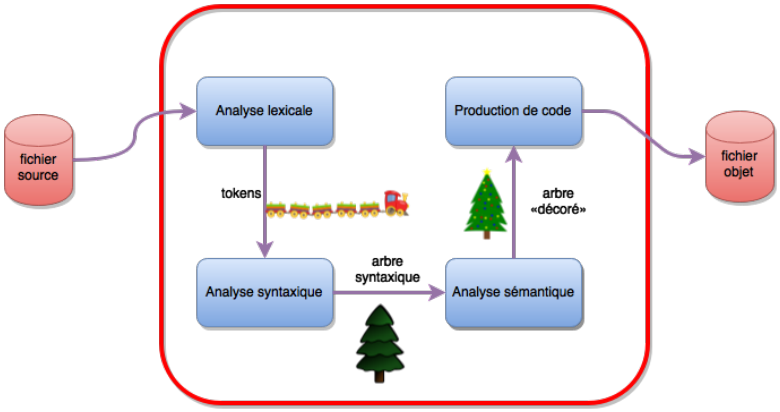
- optimisations,
- production de code …

Compilateur modulaire (2/2)

Avantage de cette architecture

- On peut avoir un compilateur avec plusieurs parties arrières pour chacune des architectures visées (partie arrière x86, x86_64, arm, …)
- Si la représentation intermédiaire est suffisamment générale,
  - *on peut avoir **n** parties avants qui produisent la forme intermédiaire*
  - *on peut avoir **m** parties arrières pour différentes architectures*
  - *Avec **n + m** parties avant/arrière on peut produire **n x m** compilateurs*
- Les compilateurs du projet GNU utilisent ce mécanisme (forme intermédiaire de type LISP **RTL**)

Principe de fonctionnement d'un compilateur



Partie avant:

- Analyseur lexical fournit des «tokens»
- Analyseur syntaxique construit un arbre
- Analyseur sémantique décore l'arbre

Partie arrière:

- Facultatif: optimisation (non représentée ici)
- Production de code parcourt l'arbre décoré

Analyse lexicale

- L'analyse lexicale consiste à découper le texte en *lexèmes* (ou *tokens*)
- élimine (éventuellement) certaines unités inutiles (espaces, commentaires)
- Dans la phrase en français:

```
La ligne comporte des mots
```

nous avons 5 lexèmes: "La", "ligne", "comporte", "des" et "mots".

Dans le programme C suivant,

```c
if (a1 == a2) a1 = a2++ + 12;
```

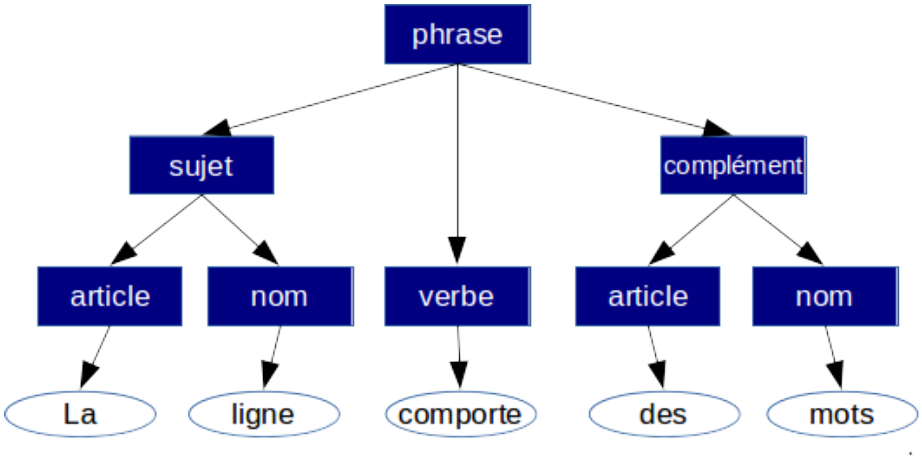
nous pouvons reconnaître les lexèmes suivants (dans le désordre)

- le mot réservé "if"
- les identificateurs "a1" et "a2"
- la constante entière 12
- les symboles "(", ")" et ";"
- les opérateurs "=" et "+"
- l'opérateur "==" (qui n'est pas vu comme deux "=" successifs)
- l'opérateur "++" (qui n'est pas vu comme deux "+" successifs)

Analyse syntaxique

- L'analyse syntaxique détermine la structure de la phrase
- Le résultat de l'analyse est un arbre syntaxique

Pour la phrase précédente, nous avons:



- Cette phrase qui est syntaxiquement correcte (puisque la suite `<sujet> <verbe> <complément>` est correcte en français).
- **Même principe pour les langages de programmation**.

Séparation analyse lexicale / syntaxique (1/2)

On peut se demander **pourquoi séparer l'analyse lexicale de l'analyse syntaxique?**

Les programmes sources contiennent beaucoup d'informations inutiles à l'analyse

- commentaires
- indentations (pas pour Python!)
- certains espaces non nécessaires `x = y ++ + 3` est équivalent à `x=y+++3`
- pour certains langages la casse n'est pas significative ("foo" ⇔ "FoO")
- la classe des grammaires nécessaires à l'analyse est généralement plus simples que celles nécessaires pour l'analyse syntaxique (langages réguliers reconnaissables par des automates finis)

Séparation analyse lexicale / syntaxique (2/2)

Pour l'**analyse syntaxique** la valeur **exacte** d'un lexème n'est pas importante.

- les affectations suivantes sont traitées de la même façon:
  - `x = y` *ou*
  - `a = b1232` *ou même*
  - `foo = 3 * bar`
  - *la règle appliquée ici est:* `<var> = <expression>`
- les classes de grammaires utilisées pour l'analyse de langages de programmation sont plus puissantes que les grammaires **rationnelles** (grammaires **algébriques** ou grammaires **non-contextuelles**)

Analyse sémantique

- L'analyse sémantique consiste à donner une signification à une phrase syntaxiquement correcte
- **La ligne comporte des mots** est juste syntaxiquement et probablement aussi sémantiquement
  - ***Des mots comportent la ligne*** *est juste syntaxiquement et probablement fausse sémantiquement*
- Pour une langue naturelle, c'est un problème très compliqué
  - ***La page comporte des phrases*** $\Longrightarrow$ *OK*
  - ***Des phrases comportent la page*** $\Longrightarrow$ *OK, pas OK?*
- Un compilateur procède à une analyse sémantique *simple* pour trouver les incohérences dans les programmes.
  - *variables non déclarés*
  - *compatibilité de types*
  - *portées des variables*
  - …

Optimisation (1/3)

- Classiquement, l'optimisation d'un programme consiste à modifier le programme pour qu'il
  - *aille plus vite*
  - *consomme moins de mémoire*
- Mais on pourrait aussi imaginer pour les applications mobiles (principalement manuel actuellement)
  - *accès réseau*
  - *consommation électrique*

L'optimisation d'un programme **ne doit pas changer la sémantique** du programme. Par exemple, la réécriture du code

```
a = 0; b = c * a;
```

par

```
a = 0; b = 0;
```

- **JUSTE** si **c** est un entier
- **FAUX** si **c** est un nombre réel (car si **c** vaut **NaN**, le produit vaut **NaN**)

Optimisation (2/3)

Les optimisations actuelles peuvent aller jusqu'à changer la nature du code utilisateur.

Par exemple, le code:

```
int fact(int num) {
  if (num > 1)
     return num * fact(num-1);
  return 1;
 }
```

compilé avec gcc-6.2 (x86_64) produit

```
# avec l'option -O1
fact(int):
        mov     eax, 1
        cmp     edi, 1
        jle     .L1
        push    rbx
        mov     ebx, edi
        lea     edi, [rdi-1]
        call    fact(int)
        imul    eax, ebx
        pop     rbx
.L1:
        rep ret
```

```
# avec l'option -O2
fact(int):
        cmp     edi, 1
        mov     eax, 1
        jle     .L2
.L1:
        imul    eax, edi
        sub     edi, 1
        cmp     edi, 1
        jne     .L1
        rep     ret
.L2:
        rep     ret
```

Optimisation (3/3)

Le code produit est donc:

```
# avec l'option -O1
fact(int):
        mov     eax, 1
        cmp     edi, 1
        jle     .L1
        push    rbx
        mov     ebx, edi
        lea     edi, [rdi-1]
        call    fact(int)
        imul    eax, ebx
        pop     rbx
.L1:
        rep ret
```

```
# avec l'option -O2
fact(int):
        cmp     edi, 1
        mov     eax, 1
        jle     .L2
.L1:
        imul    eax, edi
        sub     edi, 1
        cmp     edi, 1
        jne     .L1
        rep     ret
.L2:
        rep     ret
```

- A gauche, la traduction de la version récursive
- A droite, une **version itérative**

Conclusion:

Les compilateurs savent (souvent) mieux optimiser que le programmeur.

Production de code

Le code produit par un compilateur peut être:

- du code en langage d'assemblage (classique)
- du code pour une machine à pile fictive
  - *code simple à produire (évite les arcanes des machines réelles)*
  - *on passe souvent ensuite par un interprète de la machine à pile pour exécuter le programme*
  - *optimisations de type JIT possible*
  - *bon exemple: la JVM de Java©*
- du code dans un autre langage de programmation (souvent C)
  - *facilite la portabilité*
  - *n'est pas trop pénalisant car les compilateurs C produisent souvent du code très efficace.*
  - *permet de bénéficier des optimisations déjà existantes du langage cible (cf transparent précédent)*

Outils et théories

- Le traitement des la partie avant d'un compilateur repose sur des théories bien étudiées;
- La définition des premiers compilateurs a aussi permis de définir/concevoir de nombreuses structures de données que l'on utilise maintenant un peu partout (arbres, tables de hash, …)..

Analyse lexicale

- langages réguliers, expressions régulières
- automates
- arithmétique (reconnaissance des nombres, évaluation)

Analyse syntaxique:

- grammaires hors-contexte (**context-free grammars**)
- automates à pile
- analyseurs descendants ou ascendants
- attributs sémantiques

Analyse sémantique:

- graphes
- gestion des types
- tables des symboles, portées

Organisation du reste du cours

1. *Aspects lexicaux*
    - *Rappels sur l'outil **lex** (en TD)*
2. *Aspects syntaxiques*
    - *grammaires context-free*
    - *analyses ascendantes*
    - *analyses descendantes*
    - *l'outil **yacc***
3. *Aspects sémantiques*

    Pour étudier les aspects sémantiques, nous travaillerons sur un vrai compilateur (langage **toy**):
    - *compilateur produisant du code C*
    - *ajout de constructions simples au langage de base*
    - *ajout d'une couche objet à la Java*
    - *extensions au modèle objet de base*
4. *Optimisations de code*
    - *si on a du temps …*
