

# Compilation

## Généralités

David Teller

David.Teller@univ-orleans.fr

<http://www.univ-orleans.fr/lifo/Members/David.Teller>

03/04/2007

## Introduction

### Compilateur

### Compilation et interprétation

Compilation  
Interprétation

### Aperçu de la compilation

Précompilation  
Analyse lexicale  
Analyse syntaxique  
Analyse de types  
Optimisation  
Génération de code  
Liaison  
Assemblage

## Langages de programmation

Programmer, c'est :

1. Comprendre le problème.

## Langages de programmation

Programmer, c'est :

1. Comprendre le problème.
2. Choisir
  - ▶ un langage de programmation approprié (C ? Java ? OCaml ? Maple ?)
  - ▶ des bibliothèques de programmation appropriées (DirectX ? OpenGL ? Sdl ?)
  - ▶ des outils de développement appropriés (Visual Studio ? Eclipse ? Ant ?)

## Langages de programmation

Programmer, c'est :

1. Comprendre le problème.
2. Choisir
  - ▶ un langage de programmation approprié (C ? Java ? OCaml ? Maple ?)
  - ▶ des bibliothèques de programmation appropriées (DirectX ? OpenGL ? Sdl ?)
  - ▶ des outils de développement appropriés (Visual Studio ? Eclipse ? Ant ?)
3. Découper les sous-problèmes en modules, fonctions, structures de données.
4. Coder.



## Langages de programmation

Programmer, c'est :

1. Comprendre le problème.
2. Choisir
  - ▶ un langage de programmation approprié (C ? Java ? OCaml ? Maple ?)
  - ▶ des bibliothèques de programmation appropriées (DirectX ? OpenGL ? Sdl ?)
  - ▶ des outils de développement appropriés (Visual Studio ? Eclipse ? Ant ?)
3. Découper les sous-problèmes en modules, fonctions, structures de données.
4. Coder.
5. Déboguer.
6. Nettoyer le code, l'améliorer, le généraliser.



## Ce que vous savez faire

- Q Comment choisir un langage de programmation ?
- Q Comment choisir une bibliothèque de programmation ?
- Q Comment choisir des outils de développement ?



## Ce que vous savez faire

- Q Comment choisir un langage de programmation ?
- Q Comment choisir une bibliothèque de programmation ?
- Q Comment choisir des outils de développement ?

**A**

1. regarder ce qui existe déjà
2. regarder si vous êtes capable de vous en servir
3. vérifier les dépendances (ex. Visual Studio ⇒ Windows)
4. déterminer si cela va vous aider à résoudre le problème – ou vous gêner
5. faire l'inventaire des problèmes qui ne sont pas encore résolus



## Oui, mais...

- ▶ Et si les outils appropriés n'existent pas ?

## Oui, mais...

- ▶ Et si les outils appropriés n'existent pas ?
- ▶ Et si les bibliothèques appropriées n'existent pas ?

## Oui, mais...

- ▶ Et si les outils appropriés n'existent pas ?
- ▶ Et si les bibliothèques appropriées n'existent pas ?
- ▶ Et si le langage de programmation approprié n'existe pas ?

## Oui, mais...

- ▶ Et si les outils appropriés n'existent pas ?
- ▶ Et si les bibliothèques appropriées n'existent pas ?
- ▶ Et si le langage de programmation approprié n'existe pas ?

Dans tous ces cas, c'est à vous de les développer.

## Pourquoi créer un langage ?

**Assembleur** Parce que le langage machine est illisible.

- Ⓒ Parce que l'assembleur est incompréhensible pour des projets un peu larges.

## Pourquoi créer un langage ?

**Assembleur** Parce que le langage machine est illisible.

- Ⓒ Parce que l'assembleur est incompréhensible pour des projets un peu larges.

**Q** Ça ne suffit pas ?

## Pourquoi créer un langage ?

**Assembleur** Parce que le langage machine est illisible.

- Ⓒ Parce que l'assembleur est incompréhensible pour des projets un peu larges.

**Q** Ça ne suffit pas ?

**A** C est un langage de *programmation système*. Très adapté pour programmer le noyau Linux, pas du tout pour, mettons, écrire un jeu d'aventures, des pages web interactives ou une interface graphique.

## Pourquoi créer un langage ?

**Assembleur** Parce que le langage machine est illisible.

**C** Parce que l'assembleur est incompréhensible pour des projets un peu larges.

**Q** Ça ne suffit pas ?

**A** C est un langage de *programmation système*. Très adapté pour programmer le noyau Linux, pas du tout pour, mettons, écrire un jeu d'aventures, des pages web interactives ou une interface graphique.

**Exemples** De nos jours, Microsoft se met à *Managed C++*, Apple à Objective-C, Sun à Java, Linux à Python et C#...

## Quelques pistes

**OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.

## Quelques pistes

**OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.

**SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.

## Quelques pistes

**OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.

**SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.

**Hypercard** Parce que C est beaucoup trop compliqué pour des gens qui veulent écrire des livres interactifs.

## Quelques pistes

- OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.
- SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.
- Hypercard** Parce que C est beaucoup trop compliqué pour des gens qui veulent écrire des livres interactifs.
- Html** Parce que C est un mauvais outil pour faire de la mise en page.



## Quelques pistes

- OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.
- SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.
- Hypercard** Parce que C est beaucoup trop compliqué pour des gens qui veulent écrire des livres interactifs.
- Html** Parce que C est un mauvais outil pour faire de la mise en page.
- VB** Parce que C est beaucoup trop complexe pour des développeurs du dimanche.



## Quelques pistes

- OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.
- SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.
- Hypercard** Parce que C est beaucoup trop compliqué pour des gens qui veulent écrire des livres interactifs.
- Html** Parce que C est un mauvais outil pour faire de la mise en page.
- VB** Parce que C est beaucoup trop complexe pour des développeurs du dimanche.
- Java** Parce que C n'est pas assez sûr pour développer des applets.



## Quelques pistes

- OCaml/Haskell** Parce que C est un mauvais outil pour créer de nouveaux langages de programmation.
- SQL** Parce que C est beaucoup trop compliqué pour des secrétaires qui doivent juste enregistrer des informations.
- Hypercard** Parce que C est beaucoup trop compliqué pour des gens qui veulent écrire des livres interactifs.
- Html** Parce que C est un mauvais outil pour faire de la mise en page.
- VB** Parce que C est beaucoup trop complexe pour des développeurs du dimanche.
- Java** Parce que C n'est pas assez sûr pour développer des applets.
- ... et des milliers de langages spécialisés locaux à une entreprise, un laboratoire...



## En plus, c'est tendance

De nombreux langages sont conçus pour être *extensibles* et *adaptables* :

- ▶ Lisp/Scheme (Système des *macros*).
- ▶ OCaml (Système de préprocesseur de haut niveau Camlp4/Meta OCaml).
- ▶ Python, Ruby (Systèmes d'introspection dynamique).
- ▶ Tout ce qui rime avec XML.

## Pourquoi ce cours ?

- ▶ Pour que vous compreniez la *chaîne de développement* complète.
- ▶ Pour vous enseigner certaines techniques de conception ou/et d'extension de langage.
- ▶ Pour que vous compreniez comment fonctionnent les langages que vous utilisez déjà.

## Introduction

- Compilateur

## Compilation et interprétation

- Compilation
- Interprétation

## Aperçu de la compilation

- Précompilation
- Analyse lexicale
- Analyse syntaxique
- Analyse de types
- Optimisation
- Génération de code
- Liaison
- Assemblage

## Bilan

## Le mystère

### Au commencement

```
#include <stdio.h>
int main()
{
    printf("%s\n", "Hello_world");
    return 0;
}
```

## Le mystère

### Au commencement

```
#include <stdio.h>
int main()
{
    printf("%s\n", "Hello_world");
    return 0;
}
```

### Ensuite

```
gcc main.c -o main
```

## Le mystère

### Au commencement

```
#include <stdio.h>
int main()
{
    printf("%s\n", "Hello_world");
    return 0;
}
```

### Ensuite

```
gcc main.c -o main
```

### Enfin

```
$ ./main
Hello world
```

## Le mystère

### Au commencement

```
#include <stdio.h>
int main()
{
    printf("%s\n", "Hello_world");
    return 0;
}
```

### Ensuite

```
gcc main.c -o main
```

### Enfin

```
$ ./main
Hello world
```

Comment gcc passe-t-il de main.c à main (ou main.exe)?

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

**Analyse de types** Détecter certaines erreurs.

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

**Analyse de types** Détecter certaines erreurs.

**Optimisation** Améliorer l'arbre en conservant sa sémantique.

## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

**Analyse de types** Détecter certaines erreurs.

**Optimisation** Améliorer l'arbre en conservant sa sémantique.

**Génération de code** Transformer l'arbre en une suite d'instructions assembleur.



## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

**Analyse de types** Détecter certaines erreurs.

**Optimisation** Améliorer l'arbre en conservant sa sémantique.

**Génération de code** Transformer l'arbre en une suite d'instructions assembleur.

**Liaison** Combiner plusieurs fichiers assembleur en un seul.



## Compilation

**Précompilation** Transformer le fichier source en un autre fichier source.

**Analyse lexicale** Transformer le fichier source en une suite de termes compréhensibles par l'ordinateur.

**Analyse syntaxique** Transformer la suite de termes en un arbre de syntaxe abstraite.

**Analyse de types** Détecter certaines erreurs.

**Optimisation** Améliorer l'arbre en conservant sa sémantique.

**Génération de code** Transformer l'arbre en une suite d'instructions assembleur.

**Liaison** Combiner plusieurs fichiers assembleur en un seul.

**Assemblage** Transformer l'assembleur en langage machine.



## Au programme

### Aujourd'hui

- ▶ À propos des langages de programmation.
- ▶ Compilation vs. interprétation.
- ▶ Aperçu de chaque phase.



## Compilation

### Definition (Compilation)

La compilation est la transformation d'un programme d'un langage de programmation dans un autre.

## Compilation

### Definition (Compilation)

La compilation est la transformation d'un programme d'un langage de programmation dans un autre.

#### Exemples

- ▶ C
- ▶ C++
- ▶ C#
- ▶ Java
- ▶ OCaml
- ▶ Fortran
- ▶ JavaScript.

## Compilation native

### Definition (Compilation native)

La compilation est dite *native* si le langage de programmation final est le langage machine ou un langage lui-même compilable vers le langage machine.

**Exemples** C, C++, OCaml, Fortran.

## Compilation native

### Definition (Compilation native)

La compilation est dite *native* si le langage de programmation final est le langage machine ou un langage lui-même compilable vers le langage machine.

**Exemples** C, C++, OCaml, Fortran.

**Avantages** Exécutables optimisés.

**Inconvénients** Le compilateur est difficile à écrire, doit être réécrit pour chaque plateforme, il est difficile de faire interagir plusieurs langages.

## Compilation octet

### Definition (Compilation octet)

La compilation est dite *octet* (ou "pseudo-code machine") si le langage de programmation final doit être *interprété*.

**Exemples** Java standard (langage JVMML), C# (langage CIL)...



## Compilation octet

### Definition (Compilation chaînée)

La compilation est dite *chaînée* (ou "threaded" ou "une interprétation virtuelle") si le langage de programmation final doit être compilé – mais ne contient que des appels de fonctions d'un *moteur d'exécution*.

**Exemples** Python (compilateur PyCon), Visual Basic historique.



## Compilation octet

### Definition (Compilation octet)

La compilation est dite *octet* (ou "pseudo-code machine") si le langage de programmation final doit être *interprété*.

**Exemples** Java standard (langage JVMML), C# (langage CIL)...

**Avantages** Un seul compilateur à écrire.

**Inconvénients** Exécutable très mal optimisé.



## Compilation octet

### Definition (Compilation chaînée)

La compilation est dite *chaînée* (ou "threaded" ou "une interprétation virtuelle") si le langage de programmation final doit être compilé – mais ne contient que des appels de fonctions d'un *moteur d'exécution*.

**Exemples** Python (compilateur PyCon), Visual Basic historique.

**Avantages** Compilateur simple à écrire.

**Inconvénients** Exécutable mal optimisé.



## Interprétation

### Definition (Interpréteur)

Un *interpréteur* de code (ou "interprète") est un programme qui exécute un fichier source, en appliquant des transformations successives à l'état du système (ex. la valeur des variables en mémoire vive).

**Exemples** JavaScript, Python, Ruby, Php.

**Avantages** Beaucoup plus simple à écrire qu'un compilateur. Cycle de développement des programmes plus rapide. Un seul fichier fonctionne sur toutes les plateformes.

**Inconvénients** Exécutable très lent. Typiquement, aucune vérification avant l'exécution.

## Machine virtuelle

### Definition (Machine virtuelle)

Une machine virtuelle est un interpréteur pour un langage qui ressemble à un langage machine.

**Exemples** Java, .Net.

## Machine virtuelle

### Definition (Machine virtuelle)

Une machine virtuelle est un interpréteur pour un langage qui ressemble à un langage machine.

**Exemples** Java, .Net.

**Avantages** Un seul exécutable pour toutes les plateformes. Les exécutables sont plus sûrs que la compilation native.

**Inconvénients** Une machine virtuelle est difficile à écrire. On ne peut pas utiliser toutes les bibliothèques de la plateforme. Lent.

## Bilan

- ▶ Il existe de nombreuses techniques pour créer un langage de programmation.
- ▶ Pour ce cours, nous nous concentrerons sur la *compilation native*.

## Introduction

Compilateur

## Compilation et interprétation

Compilation

Interprétation

## Aperçu de la compilation

Précompilation

Analyse lexicale

Analyse syntaxique

Analyse de types

Optimisation

Génération de code

Liaison

Assemblage

## Bilan

## Dans notre exemple

Regardons ce que donne notre exemple.

```
cpp main.c
```

## Dans notre exemple

Regardons ce que donne notre exemple.

```
cpp main.c
```

cpp est le C Pre-Processor, le précompilateur du C.  
Après précompilation, nous disposons d'un fichier de source complet,  
dans le langage de programmation original.

## Précompilation

### Définition (Préprocesseur)

Un *préprocesseur* (ou *précompilateur*) est un programme qui transforme un fichier source en un fichier source, à l'aide de directives entrées dans un langage de programmation.

Sous gcc cpp (C Pre Processor) – le langage reconnaît les instructions `#include`, `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.

## Précompilation

### Definition (Préprocesseur)

Un *préprocesseur* (ou *précompilateur*) est un programme qui transforme un fichier source en un fichier source, à l'aide de directives entrées dans un langage de programmation.

**Sous gcc** `cpp` (C Pre Processor) – le langage reconnaît les instructions `#include`, `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.

**Sous OCaml** `camlp4` (Caml Pre-Processor Pretty Printer) – le langage de programmation est OCaml.



## Précompilation

### Definition (Préprocesseur)

Un *préprocesseur* (ou *précompilateur*) est un programme qui transforme un fichier source en un fichier source, à l'aide de directives entrées dans un langage de programmation.

**Sous gcc** `cpp` (C Pre Processor) – le langage reconnaît les instructions `#include`, `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.

**Sous OCaml** `camlp4` (Caml Pre-Processor Pretty Printer) – le langage de programmation est OCaml.

**Sous Scheme** Scheme lui-même.



## Précompilation

### Definition (Préprocesseur)

Un *préprocesseur* (ou *précompilateur*) est un programme qui transforme un fichier source en un fichier source, à l'aide de directives entrées dans un langage de programmation.

**Sous gcc** `cpp` (C Pre Processor) – le langage reconnaît les instructions `#include`, `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.

**Sous OCaml** `camlp4` (Caml Pre-Processor Pretty Printer) – le langage de programmation est OCaml.

**Sous Scheme** Scheme lui-même.

...



## Préprocesseurs lexicaux

(C, C++...)



## Préprocesseurs lexicaux

(C, C++...)

Travaillent au niveau des chaînes de caractères.



## Préprocesseurs lexicaux

(C, C++...)

Travaillent au niveau des chaînes de caractères.

- ▶ Recopier totalement un fichier dans un autre (`#include`).
- ▶ Définir des macros (`#define`).
- ▶ Remplacer le nom d'une macro par son contenu.
- ▶ Quelques tests.



## Préprocesseurs lexicaux

(C, C++...)

Travaillent au niveau des chaînes de caractères.

- ▶ Recopier totalement un fichier dans un autre (`#include`).
- ▶ Définir des macros (`#define`).
- ▶ Remplacer le nom d'une macro par son contenu.
- ▶ Quelques tests.

Langage de préprocesseur extrêmement limité mais indispensable.



## Préprocesseurs syntaxiques

(OCaml, Scheme...)



## Préprocesseurs syntaxiques

(OCaml, Scheme...)

Travaillent au niveau des arbres syntaxiques.



## Préprocesseurs syntaxiques

(OCaml, Scheme...)

Travaillent au niveau des arbres syntaxiques.

Appliquer des fonctions au programme pour le transformer avant de le compiler.



## Préprocesseurs syntaxiques

(OCaml, Scheme...)

Travaillent au niveau des arbres syntaxiques.

Appliquer des fonctions au programme pour le transformer avant de le compiler.

Langage de préprocesseur puissant et extensible.



## Langages réflexifs

(Python, Ruby, Javascript, Scheme...)



## Langages réflexifs

(Python, Ruby, Javascript, Scheme...)  
Travaillent au niveau des arbres syntaxiques.



## Langages réflexifs

(Python, Ruby, Javascript, Scheme...)  
Travaillent au niveau des arbres syntaxiques.  
Appliquer des fonctions au programme pour le transformer pendant son exécution.



## Langages réflexifs

(Python, Ruby, Javascript, Scheme...)  
Travaillent au niveau des arbres syntaxiques.  
Appliquer des fonctions au programme pour le transformer pendant son exécution.  
Langage puissant et extensible.



## Analyse lexicale

### Definition (Analyseur lexical)

Un *analyseur lexical* (ou *lexeur* ou *scanner* ou *reconnaisseur*) est une fonction qui rassemble les caractères consécutifs en *lexèmes* (ou "mots" ou *jeton* ou *token*).



## Analyse lexicale

### Definition (Analyseur lexical)

Un *analyseur lexical* (ou *lexeur* ou *scanner* ou *reconnaisseur*) est une fonction qui rassemble les caractères consécutifs en *lexèmes* (ou "mots" ou *jeton* ou *token*).

En français, l'analyse lexicale revient à

- ▶ ignorer les espaces et la mise en page ;
- ▶ rassembler les lettres en mots, les chiffres en nombres. ... ;
- ▶ reconnaître la ponctuation ;
- ▶ ...



## Analyse lexicale

### Definition (Analyseur lexical)

Un *analyseur lexical* (ou *lexeur* ou *scanner* ou *reconnaisseur*) est une fonction qui rassemble les caractères consécutifs en *lexèmes* (ou "mots" ou *jeton* ou *token*).

En français, l'analyse lexicale revient à

- ▶ ignorer les espaces et la mise en page ;
- ▶ rassembler les lettres en mots, les chiffres en nombres. ... ;
- ▶ reconnaître la ponctuation ;
- ▶ ...

Mots, nombres, ponctuation...sont les lexèmes.



## Exemple

Partons d'une chaîne de caractères.

```
"int x = 1024 * some_y /*with comments*/;"
```



## Exemple

Partons d'une chaîne de caractères.

```
"int x = 1024 * some_y /*with comments*/;"
```

...et obtenons une liste de lexèmes

```
TYPE, ID, EQUALS, INT_VALUE, OPERATOR, ID, END_OF_SENTENCE
```



## Exemple

Partons d'une chaîne de caractères.

```
"int x = 1024 * some_y /*with comments*/;"
```

...et obtenons une liste de lexèmes

```
TYPE, ID, EQUALS, INT_VALUE, OPERATOR, ID, END_OF_SENTENCE
```

...chaque lexème contient des informations supplémentaires

```
TYPE, ID, EQUALS, INT_VALUE, OPERATOR, ID, END_OF_SENTENCE
"int" "x"          1024      *      "some_y"
```

## Fonctionnement

L'analyseur lexical est typiquement défini par un ensemble de règles de la forme :

- ▶ "tout caractère entre '0' et '9' est un chiffre"
- ▶ "toute succession de chiffres est un nombre"
- ▶ "toute succession de caractères entre 'a' et 'z' ou entre 'A' et 'Z' est un nom"
- ▶ "toute succession de caractères entre /\* et \*/ doit être ignorée"

## Fonctionnement

L'analyseur lexical est typiquement défini par un ensemble de règles de la forme :

- ▶ "tout caractère entre '0' et '9' est un chiffre"
- ▶ "toute succession de chiffres est un nombre"
- ▶ "toute succession de caractères entre 'a' et 'z' ou entre 'A' et 'Z' est un nom"
- ▶ "toute succession de caractères entre /\* et \*/ doit être ignorée"

Ces règles définissent des *expressions régulières*.

## Résultat

Avec l'outil Flex, on écrira par exemple

```
#token INT_VALUE "[0-9]+"
```

```
#token ID "[a-zA-Z]+"
```

## Résultat

Avec l'outil Flex, on écrira par exemple

```
#token INT_VALUE "[0-9]+"
```

```
#token ID "[a-zA-Z]+"
```

Après analyse lexicale, nous avons une liste (ou un flux) de lexèmes, de type INT\_VALUE, ID, OP...

## Analyseur syntaxique

### Definition (Analyseur syntaxique)

Un *analyseur syntaxique* (ou *parseur*) est une fonction qui transforme une suite de *lexèmes* en un *arbre de syntaxe abstraite*.

## Analyseur syntaxique

### Definition (Analyseur syntaxique)

Un *analyseur syntaxique* (ou *parseur*) est une fonction qui transforme une suite de *lexèmes* en un *arbre de syntaxe abstraite*.

En français, l'analyse syntaxique revient à

- ▶ reconnaître sujet, verbe, complément ;
- ▶ reconnaître les propositions ;
- ▶ reconnaître les phrases ;
- ▶ reconnaître les titres ;
- ▶ ...

## Analyseur syntaxique

### Definition (Analyseur syntaxique)

Un *analyseur syntaxique* (ou *parseur*) est une fonction qui transforme une suite de *lexèmes* en un *arbre de syntaxe abstraite*.

En français, l'analyse syntaxique revient à

- ▶ reconnaître sujet, verbe, complément ;
- ▶ reconnaître les propositions ;
- ▶ reconnaître les phrases ;
- ▶ reconnaître les titres ;
- ▶ ...

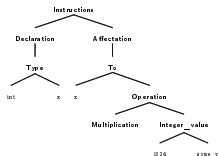
Sujet, verbe, complément, propositions... sont des arbres de syntaxe abstraite.

## Exemple

En partant d'une liste de lexèmes  
 $\underbrace{\text{TYPE, ID, EQUALS, INT\_VALUE, OPERATOR}}_{\text{"int" "x" 1024 "MULT"}}, \underbrace{\text{ID}}_{\text{"some\_y"}}, \text{END\_OF\_SENTENCE}$

## Exemple

En partant d'une liste de lexèmes  
 $\underbrace{\text{TYPE, ID, EQUALS, INT\_VALUE, OPERATOR}}_{\text{"int" "x" 1024 "MULT"}}, \underbrace{\text{ID}}_{\text{"some\_y"}}, \text{END\_OF\_SENTENCE}$   
 ...on obtient l'arbre syntaxique suivant



## Fonctionnement

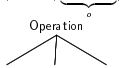
L'analyseur syntaxique est lui aussi défini par un ensemble de règles de la forme :

- ▶ "un nombre  $N$  peut être transformé en arbre syntaxique  $\text{Integer\_value}$



, qui est une Expression"

- ▶ "si  $A$  et  $B$  peuvent être transformés en deux arbres syntaxiques  $\text{Arbre}_A$  et  $\text{Arbre}_B$ , qui sont des Expressions, alors  $A, \text{OPERATOR}, B$



peut être transformé en un arbre syntaxique  $o$

- ▶ ...

## Fonctionnement

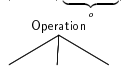
L'analyseur syntaxique est lui aussi défini par un ensemble de règles de la forme :

- ▶ "un nombre  $N$  peut être transformé en arbre syntaxique  $\text{Integer\_value}$



, qui est une Expression"

- ▶ "si  $A$  et  $B$  peuvent être transformés en deux arbres syntaxiques  $\text{Arbre}_A$  et  $\text{Arbre}_B$ , qui sont des Expressions, alors  $A, \text{OPERATOR}, B$



peut être transformé en un arbre syntaxique  $o$

- ▶ ...

Ces règles définissent une *grammaire*.

## Résultat

Avec l'outil Bison, on écrira quelque chose du genre

```
expression :  
| expression OPERATOR expression  
| INTEGER_VALUE  
|
```



## Types

### Definition (Type)

Le *type* d'une valeur est une information sur cette valeur qui détermine comment elle peut être manipulée, quelles fonctions et opérateurs peuvent lui être appliquées, et comment elle peut interagir avec d'autres valeurs.

### Exemples

- ▶ 1024 est un *entier*;
- ▶ "truc" est une *chaîne de caractères*;
- ▶ ...



## Résultat

Avec l'outil Bison, on écrira quelque chose du genre

```
expression :  
| expression OPERATOR expression  
| INTEGER_VALUE  
|
```

Après analyse syntaxique, nous avons un *arbre*, que nous pouvons parcourir.



## Vérification de types

### Definition (Vérification de types)

La *vérification de types* est une forme d'analyse de l'arbre syntaxique qui vérifie que opérations sur les valeurs sont compatibles avec la définition de ces valeurs.

### Exemples

- ▶  $1024 * 2$  est une opération licite car  $*$  peut s'appliquer à deux entiers;
- ▶ dans la majorité des langages de programmation "truc"+2 n'a aucun sens;
- ▶ float c = "test" n'a aucun sens;
- ▶ si f est une fonction qui prend en argument un float et si on appelle f(x), x a intérêt à être un float.



## Niveaux de vérification

**Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.



## Niveaux de vérification

**Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.

**C** Presque aucune vérification. Vous pouvez écrire presque n'importe quoi, ça s'exécutera – et ça plantera *sans* message d'erreur.



## Niveaux de vérification

**Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.

**C** Presque aucune vérification. Vous pouvez écrire presque n'importe quoi, ça s'exécutera – et ça plantera *sans* message d'erreur.

**OCaml** Nombreuses vérifications sur la sûreté d'exécution. Il est difficile de faire planter un programme.



## Niveaux de vérification

**Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.

**C** Presque aucune vérification. Vous pouvez écrire presque n'importe quoi, ça s'exécutera – et ça plantera *sans* message d'erreur.

**OCaml** Nombreuses vérifications sur la sûreté d'exécution. Il est difficile de faire planter un programme.

**Camelot** Vérifications sur la sûreté d'exécution, sur la quantité de mémoire vive utilisée, sur la désallocation de mémoire, etc.



## Niveaux de vérification

- Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.
- C** Presque aucune vérification. Vous pouvez écrire presque n'importe quoi, ça s'exécutera – et ça plantera *sans* message d'erreur.
- OCaml** Nombreuses vérifications sur la sûreté d'exécution. Il est difficile de faire planter un programme.
- Camelot** Vérifications sur la sûreté d'exécution, sur la quantité de mémoire vive utilisée, sur la désallocation de mémoire, etc.
- Coq** *Tout* est vérifié. Chaque programme est accompagné d'une preuve mathématique de *toutes* ses propriétés intéressantes.

## Niveaux de vérification

- Python** Très exactement aucune vérification. Vous pouvez écrire n'importe quoi, ça s'exécutera – et ça plantera avec un message d'erreur.
- C** Presque aucune vérification. Vous pouvez écrire presque n'importe quoi, ça s'exécutera – et ça plantera *sans* message d'erreur.
- OCaml** Nombreuses vérifications sur la sûreté d'exécution. Il est difficile de faire planter un programme.
- Camelot** Vérifications sur la sûreté d'exécution, sur la quantité de mémoire vive utilisée, sur la désallocation de mémoire, etc.
- Coq** *Tout* est vérifié. Chaque programme est accompagné d'une preuve mathématique de *toutes* ses propriétés intéressantes.

Pour C, il existe des vérificateurs externes, tels que Uno.

## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

- ▶ comme 1024 et 2 sont des entiers et comme \* est une opération qui, à partir de deux entiers construit un entier,  $1024 * 2$  est un entier

## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

- ▶ comme 1024 et 2 sont des entiers et comme \* est une opération qui, à partir de deux entiers construit un entier,  $1024 * 2$  est un entier
- ▶ si  $f$  est une fonction qui produit des float,  $f(x)$  est forcément un float



## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

- ▶ comme 1024 et 2 sont des entiers et comme \* est une opération qui, à partir de deux entiers construit un entier,  $1024 * 2$  est un entier
  - ▶ si  $f$  est une fonction qui produit des float,  $f(x)$  est forcément un float
- C Presque aucune inférence. Presque tout doit être précisé par le programmeur.



## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

- ▶ comme 1024 et 2 sont des entiers et comme \* est une opération qui, à partir de deux entiers construit un entier,  $1024 * 2$  est un entier
- ▶ si  $f$  est une fonction qui produit des float,  $f(x)$  est forcément un float

C Presque aucune inférence. Presque tout doit être précisé par le programmeur.

OCaml Presque tout est déterminé automatiquement par le compilateur.



## Inférence de types

### Definition (Inférence de types)

L'inférence de types est une phase d'analyse de l'arbre syntaxique qui détermine, pour chaque valeur, son type.

- ▶ comme 1024 et 2 sont des entiers et comme \* est une opération qui, à partir de deux entiers construit un entier,  $1024 * 2$  est un entier
- ▶ si  $f$  est une fonction qui produit des float,  $f(x)$  est forcément un float

C Presque aucune inférence. Presque tout doit être précisé par le programmeur.

OCaml Presque tout est déterminé automatiquement par le compilateur.

Coq Inférence interactive : le compilateur trouve des choses, pose des questions, trouve d'autres choses, etc.



## Résultat

Après la phase d'analyse de types

- ▶ l'arbre est étiqueté avec des informations de types
- ▶ le source est accepté ou refusé
- ▶ (idéalement) on garantit qu'il ne peut pas y avoir d'erreurs dans les types – un `int` contiendra toujours un entier, etc.



## Optimisation

### Definition (Optimisation)

Jeu de transformations sur l'arbre syntaxique, qui supprime les opérations inutiles.



## Résultat

Après la phase d'analyse de types

- ▶ l'arbre est étiqueté avec des informations de types
- ▶ le source est accepté ou refusé
- ▶ (idéalement) on garantit qu'il ne peut pas y avoir d'erreurs dans les types – un `int` contiendra toujours un entier, etc.

En pratique, les analyseurs de types sont souvent faits par des gens qui n'y connaissent rien – et ne garantissent donc souvent rien. Mais on fait comme si.



## Optimisation

### Definition (Optimisation)

Jeu de transformations sur l'arbre syntaxique, qui supprime les opérations inutiles.

#### Exemple

```
"int x = 1; y = x*10;" – on peut remplacer directement par y = 10  
if(0) printf("%s", "Test \n"); ne fait rien – on peut donc  
supprimer tout cela.
```



## Catégories d'optimisations

Quelques techniques d'optimisation :

- ▶ Remplacer une suite d'instructions par une instruction équivalent mais connue comme plus rapide.
- ▶ Déterminer comment une fonction est utilisée de manière à améliorer le code de cette fonction.
- ▶ Remplacer un appel de fonction par le code source de la fonction.
- ▶ Remplacer une boucle de longueur constante par la suite d'instructions équivalente.
- ▶ Remplacer une boucle par des instructions exécutées en même temps sur plusieurs processeurs.
- ▶ Sortir du code commun hors d'une boucle.
- ▶ ...



## Catégories d'optimisations

Quelques techniques d'optimisation :

- ▶ Remplacer une suite d'instructions par une instruction équivalent mais connue comme plus rapide.
- ▶ Déterminer comment une fonction est utilisée de manière à améliorer le code de cette fonction.
- ▶ Remplacer un appel de fonction par le code source de la fonction.
- ▶ Remplacer une boucle de longueur constante par la suite d'instructions équivalente.
- ▶ Remplacer une boucle par des instructions exécutées en même temps sur plusieurs processeurs.
- ▶ Sortir du code commun hors d'une boucle.
- ▶ ...

(Presque) toutes ces optimisations sont des transformations sur l'arbre syntaxique.



## Résultat

Après la phase d'optimisation, nous avons un arbre du même genre que l'arbre précédent.



## Génération de code

### Definition (Génération de code)

La *génération de code* est une opération qui transforme un arbre de syntaxe abstraite en un fichier de code source d'un langage de bas niveau.



## Génération de code

### Definition (Génération de code)

La *génération de code* est une opération qui transforme un arbre de syntaxe abstraite en un fichier de code source d'un langage de bas niveau.

Généralement, le langage de bas niveau est

1. l'*assembleur*
2. le C
3. l'assembleur virtuel .Net
4. l'assembleur virtuel Java.

## Exemple

```
gcc -S -c main.c
```

## Génération de code

### Definition (Génération de code)

La *génération de code* est une opération qui transforme un arbre de syntaxe abstraite en un fichier de code source d'un langage de bas niveau.

Généralement, le langage de bas niveau est

1. l'*assembleur*
2. le C
3. l'assembleur virtuel .Net
4. l'assembleur virtuel Java.

Dans tous les cas, il s'agit d'un langage de plus bas niveau.

## Fonctionnement

Le processus de compilation est déterminé par un ensemble de règles telles que :

"Si  $x$  est un entier qui apparaît en mémoire vive à l'adresse  $a_x$ , si  $y$  est un entier qui apparaît en mémoire vive à l'adresse  $a_y$  et si  $z$  est un entier qui apparaît en mémoire vive à l'adresse  $a_z$ , alors pour calculer  $z = x \times y$ , la procédure consiste à

1. charger le contenu de  $a_x$  dans le processeur (registre  $r_1$ )
2. charger le contenu de  $a_y$  dans le processeur (registre  $r_2$ )
3. invoquer l'instruction processeur `imul  $r_1, r_2, r_3$`
4. placer le contenu du registre  $r_3$  à l'adresse mémoire  $a_z$ ."

## Fonctionnement

Le processus de compilation est déterminé par un ensemble de règles telles que :

"Si  $x$  est un entier qui apparaît en mémoire vive à l'adresse  $a_x$ , si  $y$  est un entier qui apparaît en mémoire vive à l'adresse  $a_y$  et si  $z$  est un entier qui apparaît en mémoire vive à l'adresse  $a_z$ , alors pour calculer  $z = x \times y$ , la procédure consiste à

1. charger le contenu de  $a_x$  dans le processeur (registre  $r_1$ )
2. charger le contenu de  $a_y$  dans le processeur (registre  $r_2$ )
3. invoquer l'instruction processeur `imul r1, r2, r3`
4. placer le contenu du registre  $r_3$  à l'adresse mémoire  $a_z$ ."

Il s'agit d'autant de règles de *filtrage par motifs* lors d'un *parcours d'arbre en profondeur*.



## Fragments additionnels.

" Typiquement, il est aussi nécessaire d'ajouter quelques fragments de code constants

**Le prélude** ajouté au début du code et qui précise si le fichier est un exécutable ou une bibliothèque, l'enregistre auprès du système...

**La postlude** ajouté à la fin du code et qui nettoie tout ce qui peut être nettoyé.



## Résultat

À l'issue de la phase de génération de code, nous avons un code source dans un langage de bas niveau.



## Résultat

À l'issue de la phase de génération de code, nous avons un code source dans un langage de bas niveau.  
 Tout ceci suffit, lorsque notre programme utilise un seul fichier.



## Liaison

### Definition (Éditeur de lien)

L'*éditeur de lien* (ou *lieur*) combine plusieurs fichiers source de bas niveau en un seul.



## Fonctionnement

Le lieur procède généralement de la manière suivante :

- ▶ faire la liste de tous les *symboles* (variables et fonctions) définis dans chaque fichier source
- ▶ faire la liste de tous les symboles utilisés dans chaque fichier source
- ▶ recopier tous les symboles qui sont utilisés vers un fichier source commun et conserver la trace de l'*adresse* de chaque symbole après copie
- ▶ remplacer les *références* à chaque symbole par l'adresse du symbole.



## Liaison

### Definition (Éditeur de lien)

L'*éditeur de lien* (ou *lieur*) combine plusieurs fichiers source de bas niveau en un seul.

### Exemple

```
gcc -S main.c
```



## Fonctionnement

Le lieur procède généralement de la manière suivante :

- ▶ faire la liste de tous les *symboles* (variables et fonctions) définis dans chaque fichier source
- ▶ faire la liste de tous les symboles utilisés dans chaque fichier source
- ▶ recopier tous les symboles qui sont utilisés vers un fichier source commun et conserver la trace de l'*adresse* de chaque symbole après copie
- ▶ remplacer les *références* à chaque symbole par l'adresse du symbole.

**Note** C'est aussi ce que fait Mac OS X lors de sa phase d'*optimisation*, après l'installation de nouveaux programmes.



## Résultat

Après liaison, nous avons un fichier source complet.



## Assemblage

### Definition (Assembleur)

L'assembleur est un ensemble de notations (à peu près) lisibles par un être humain pour le langage machine, c'est-à-dire pour l'ensemble des commandes connus directement par le microprocesseur.

Il existe un assembleur différent par micro-processeur. En particulier, l'assembleur Pentium MMX II contient l'assembleur Pentium MMX, qui contient l'assembleur Pentium, qui contient l'assembleur 486...



## Assemblage

### Definition (Assembleur)

L'assembleur est un ensemble de notations (à peu près) lisibles par un être humain pour le langage machine, c'est-à-dire pour l'ensemble des commandes connus directement par le microprocesseur.



## Assemblage

### Definition (Assembleur)

L'assembleur est un ensemble de notations (à peu près) lisibles par un être humain pour le langage machine, c'est-à-dire pour l'ensemble des commandes connus directement par le microprocesseur.

Il existe un assembleur différent par micro-processeur. En particulier, l'assembleur Pentium MMX II contient l'assembleur Pentium MMX, qui contient l'assembleur Pentium, qui contient l'assembleur 486...

### Definition (Assemblage)

La phase d'assemblage consiste à remplacer les notations assembleur par les notations correspondantes en langage machine.



## Fonctionnement

Le leur dispose d'une suite de règles simples :

- ▶ l'instruction `imul` se traduit par `1111`
- ▶ l'instruction `push` se traduit par ...
- ▶ ...



## Résultat

À la fin de la phase d'assemblage, nous avons (enfin) un exécutable ou une bibliothèque utilisable par le système !



## Fonctionnement

Le leur dispose d'une suite de règles simples :

- ▶ l'instruction `imul` se traduit par `1111`
- ▶ l'instruction `push` se traduit par ...
- ▶ ...

La phase d'assemblage est la plus simple, puisque chaque commande assembleur correspond à une suite connue de 0s et de 1s.



Introduction  
Compilateur

Compilation et interprétation  
Compilation  
Interprétation

Aperçu de la compilation  
Précompilation  
Analyse lexicale  
Analyse syntaxique  
Analyse de types  
Optimisation  
Génération de code  
Liaison  
Assemblage

Bilan



## Où en sommes-nous ?

Nous avons vu un

- ▶ le rôle de la compilation
- ▶ un aperçu de chaque phase de la compilation.

## Pour la suite

Nous allons regarder plus en détails :

- ▶ l'analyse lexicale
- ▶ l'analyse syntaxique
- ▶ la génération de code.