

Introduction au cours de compilation

Option INFO — TLANG

Pierre Molinaro

Novembre 2019

Introduction

But de l'enseignement « TLANG »

Donner des connaissances théoriques et pratiques sur la compilation :

Connaissances théoriques : analyse lexicale, analyse syntaxique ;

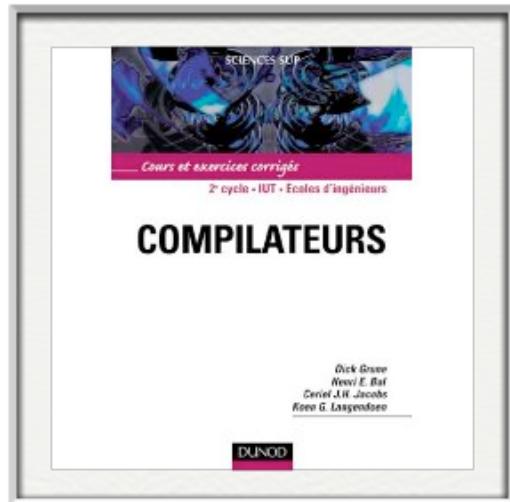
Connaissances pratiques :

- Lex & Yacc, GCC, LLVM ;
- utilisation d'un générateur d'analyseur lexical (lex), syntaxique (Yacc) ;
- utilisation d'un générateur de compilateur (GALGAS) ;

Planning (prévisionnel)

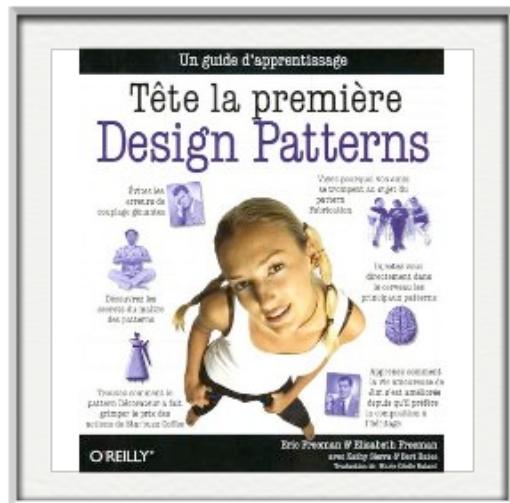
Cours	Introduction	2 h	Pierre Molinaro
	Analyse lexicale	5 h	Didier Lime
	Analyse syntaxique	5 h	Didier Lime
	Machines de Turing	2 h	Didier Lime
	LLVM et génération de code	4 h	Pierre Molinaro
TP	Lex-Yacc	2 h	Didier Lime
	GALGAS	10 h	Pierre Molinaro

Bibliographie



[1] Compilateurs : cours et exercices corrigés

Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen,
traduction de C. Fédèle et O. Lecarme
Éditions DUNOD, ISBN 2 10 005887 8

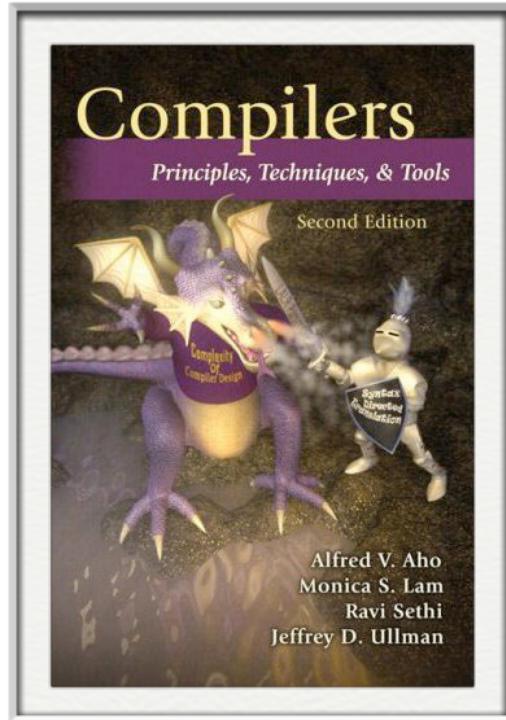


[2] Design Patterns : Tête la première

Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates,
traduction de Marie-Cécile Baland
Éditions O'Reilly France, ISBN : 2 84177 350 7

Voir aussi : <http://www.design-patterns.fr/>

Bibliographie



The Dragon Book

Compilers : Principles, Techniques and Tools

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman

Pearson Editions

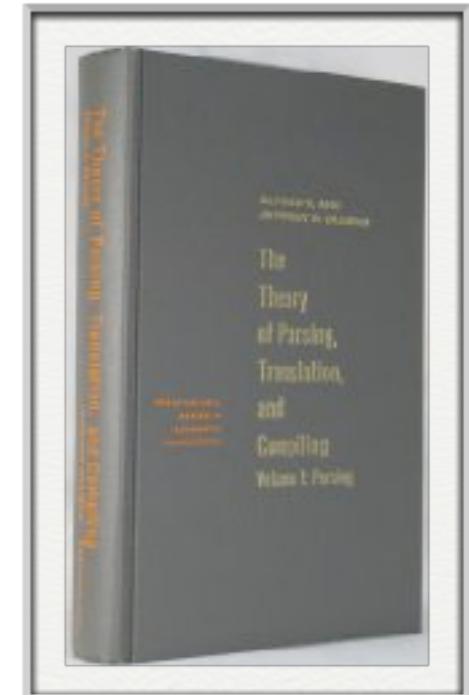
The Theory of Parsing, Translation and Compiling

Volume I: Parsing (1972)

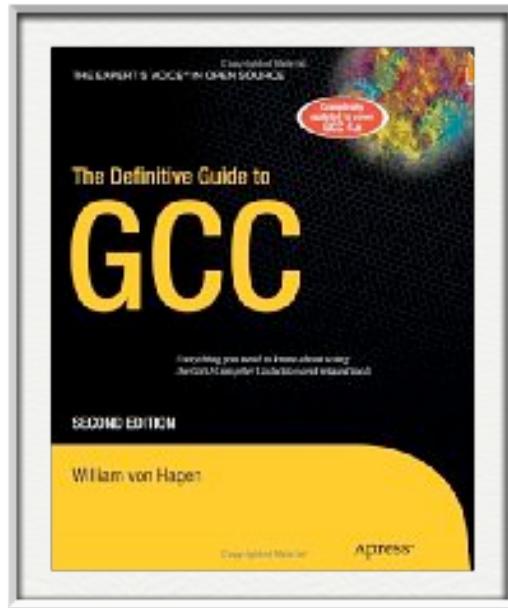
Volume II: Compiling (1973)

Alfred V. Aho, Jeffrey D. Ullman

Prentice-Hall



Bibliographie

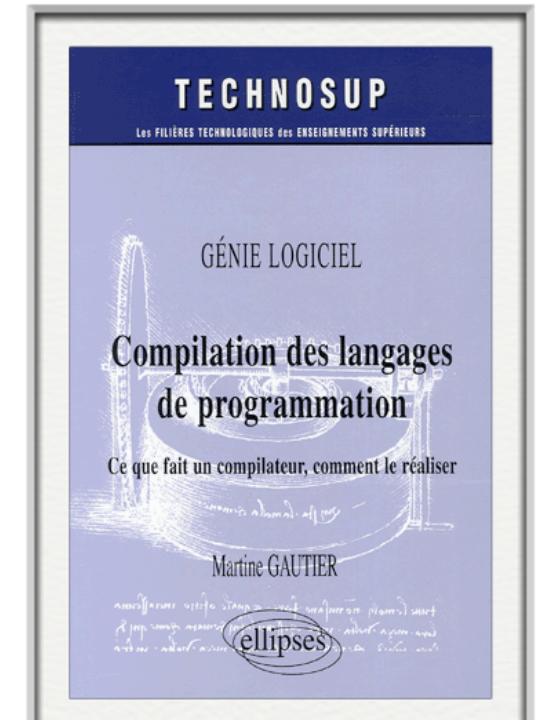


[3] The definitive Guide to GCC, Second Edition

William von Hagen,
Éditions Apress

[4] Compilation des langages de programmation

Martine Gautier
Éditions Ellipses, ISBN 2 7298 2620 3



Que fait un compilateur ?

Compilateur : programme transformationnel qui accepte en entrée un programme rédigé dans un langage (dit *langage source*) et qui fournit en sortie sa traduction dans un autre langage (dit *langage objet*).

Souvent :

- le langage source est un texte ;
- le langage objet est binaire.

Besoins actuels nouveaux :

- coloration syntaxique ;
- compilateurs « just in time » ;
- messages d'erreurs avec proposition de correction ;
- outils de transformation de source.

Historique

Le tout début. Le langage machine (très fastidieux).

Un progrès (1950) : l'assembleur (attention, en français *assembleur* signifie aussi bien le langage (*assembly language*), que le traducteur (*assemble*) des textes sources en code objet). Utilisation de mnémoniques, d'étiquettes. Problème : le code est complètement à réécrire pour une autre machine.

Apparition des premiers compilateurs : Fortran (à partir de 1957). L'accent était mis sur la qualité des optimisations du code engendré.

Avancées théoriques : les grammaires (étudiées dans les années 1950 / 1970) ont permis de formaliser une partie de la conception des compilateurs, concrétisées dans les langages tels que Pascal (Niklaus Wirth, 1970, [http://fr.wikipedia.org/wiki/Pascal_\(langage\)](http://fr.wikipedia.org/wiki/Pascal_(langage))) et Simula 67 (Daal & Nigaard, 1967, <http://en.wikipedia.org/wiki/Simula>).

Actuellement. Engendrer du code machine efficace est compliqué. Donc, il n'existe qu'un nombre restreint de compilateurs engendant du code machine. Il est plus simple d'engendrer du code dans un langage généraliste (C, Java, ...), ou un code intermédiaire tel que LLVM.

Les « Domain Specific Languages »

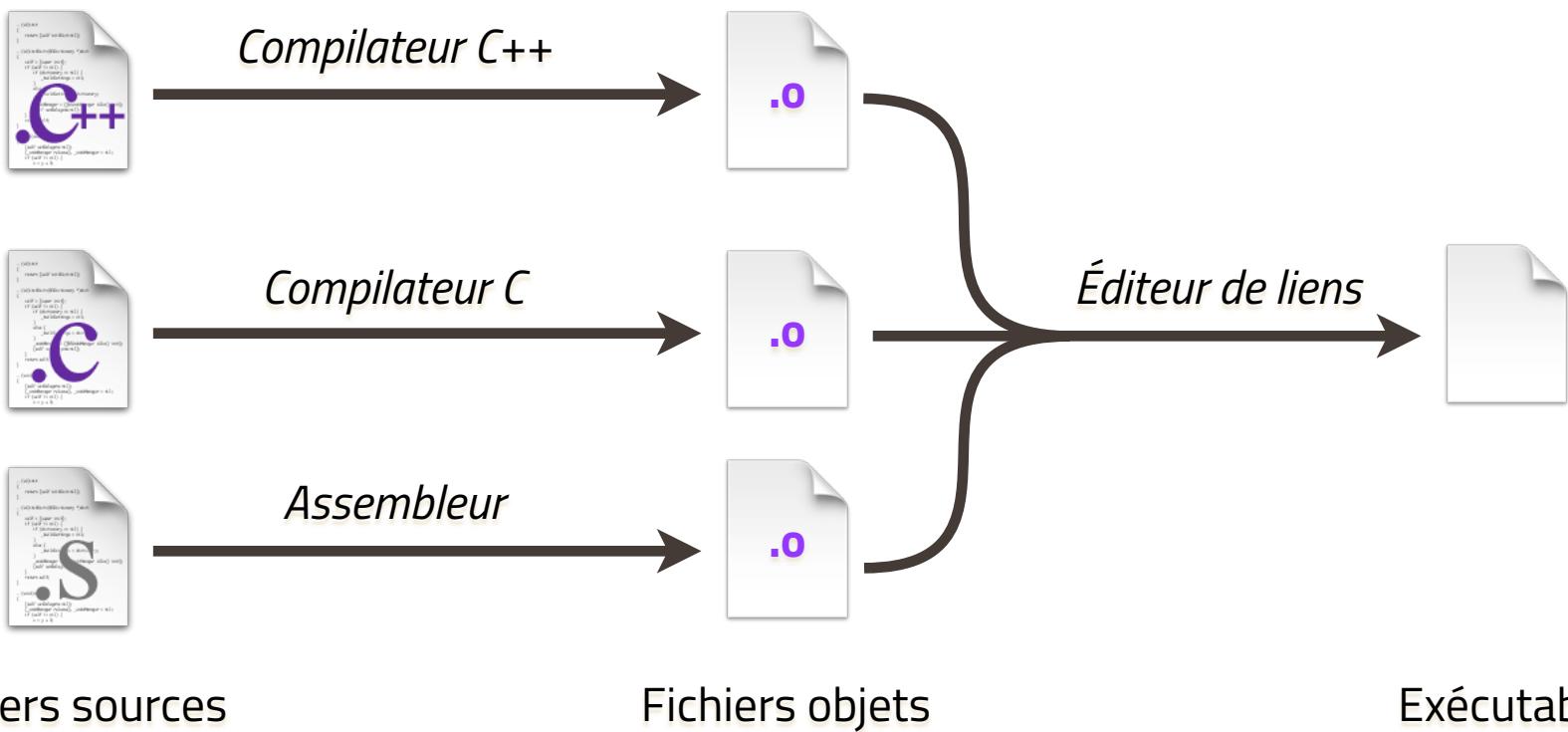
(http://en.wikipedia.org/wiki/Domain-specific_language).

Par opposition aux langages à usage général (C, C++, Java, ...), ils sont dédiés à une classe de problèmes : Lex, Yacc, AltaRica (<http://altarica.labri.fr/wiki/>), Spice (<http://bwrc.eecs.berkeley.edu/classes/icbook/spice/>)... Leur spécialisation permet de concevoir des analyseurs sémantiques capables de vérifier des propriétés sur les textes sources.

Propriétés d'un compilateur

- Pouvoir traiter des sources d'une taille arbitraire ;
- Ne pas échouer silencieusement ;
- Continuer après détection d'une erreur ;
- Robuste (ne planter brutalement après détection d'une erreur) ;
- Afficher des messages d'erreurs clairs ;
- Compilation séparée (ce qui implique un éditeur de liens) ;
- Rapidité ;
- Qualité du code engendré ;

Schéma général — compilation et édition de liens (exemple typique)



Un fichier objet a toujours la même structure, quelque soit sa provenance.

Familles de compilateurs

Problème : faciliter l'écriture d'une famille de compilateurs :

- qui accepte plusieurs langages sources ;
- qui engendre du code pour différentes cibles.

Définition d'une représentation intermédiaire :

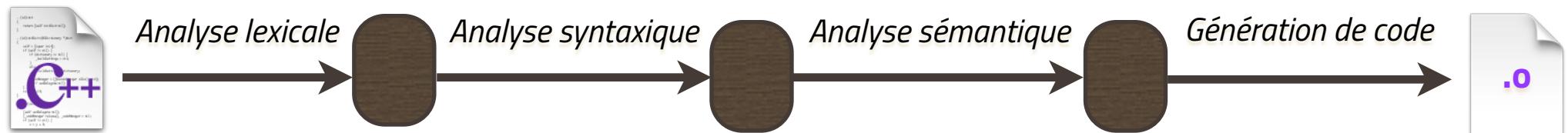
- indépendante du langage source ;
- indépendante de la machine cible.

Le compilateur est alors « divisé » en deux compilateurs qui sont successivement appelés :

- le *front-end*, qui traduit un programme source dans sa représentation intermédiaire ;
- le *back-end*, qui traduit la représentation intermédiaire en code machine.

Les différentes phases de la compilation

Les différentes phases de la compilation



Symbolise une structure de données en mémoire

La phase de génération de code *peut* inclure une optimisation.

L'analyse lexicale

L'analyse lexicale regroupe les caractères du fichier source en *terminaux syntaxiques*, ou encore *tokens*, en éliminant séparateurs et commentaires.

Par exemple :

```
int main (void) {
    printf ("Hello world\n") ;
    return 0 ;
}
```



```
int
identificateur : "main"
(
void
)
{
identificateur : "printf"
(
chaîne : "Hello world\n"
)
;
return
nombre : "0"
;
}
```

Les outils théoriques sont les *automates d'états finis* et les *grammaires régulières* (cours de Didier Lime).

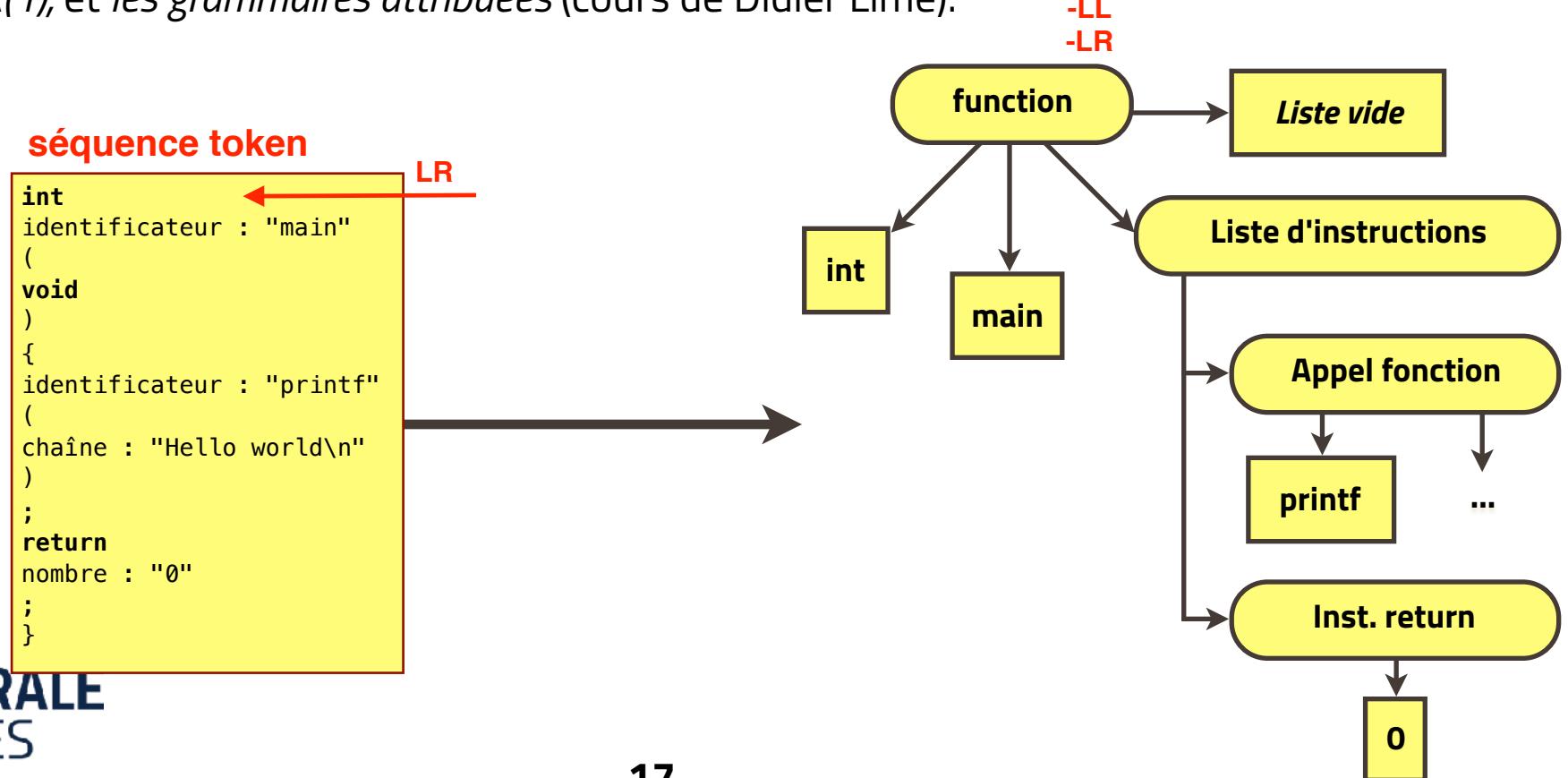
L'analyse syntaxique

L'analyse syntaxique a un double but :

- vérifier le *bon séquencement* des éléments terminaux ;
- de construire l'*arbre syntaxique abstrait (AST : Abstract Syntax Tree)* du texte source.

Les outils théoriques sont les *grammaires hors contexte*, notamment les propriétés *LL(1)* et *LR(1)*, et les *grammaires attribuées* (cours de Didier Lime).

type de grammar
-LL
-LR



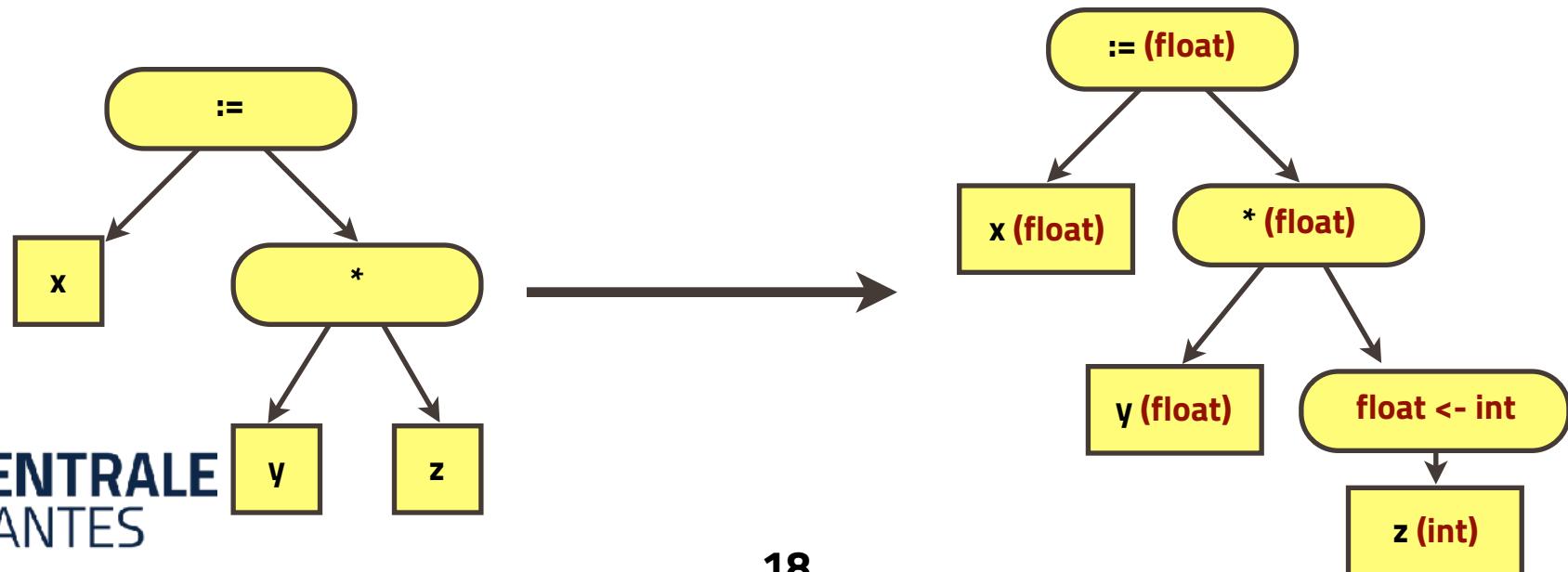
L'analyse sémantique

L'analyse sémantique inclut la vérification des types, de la déclaration correcte des variables... Cette analyse applique la *sémantique statique* (par opposition à la *sémantique dynamique*, qui est la génération de code ou son exécution).

Exemple : ces deux instructions C ont la même sémantique statique :

```
if (a == 0) { ... }  
while (a == 0) { ... }
```

L'arbre syntaxique abstrait est annoté / décoré pour préciser la sémantique, grâce à des informations sémantiques contextuelles (par exemple : x et y sont des flottants, et z un entier).



La génération de code (1/2)

Le code engendré peut être de différentes natures :

- directement du code exécutable ;
- du code objet contenant du code machine, prêt à être lié à d'autres codes objet ;
- du code objet contenant du code virtuel (exemple : *bytecode Java*, http://fr.wikipedia.org/wiki/Bytecode_Java) ;
- du texte (source C, C++, ...) ;
- du code intermédiaire (LLVM) ;
- ...

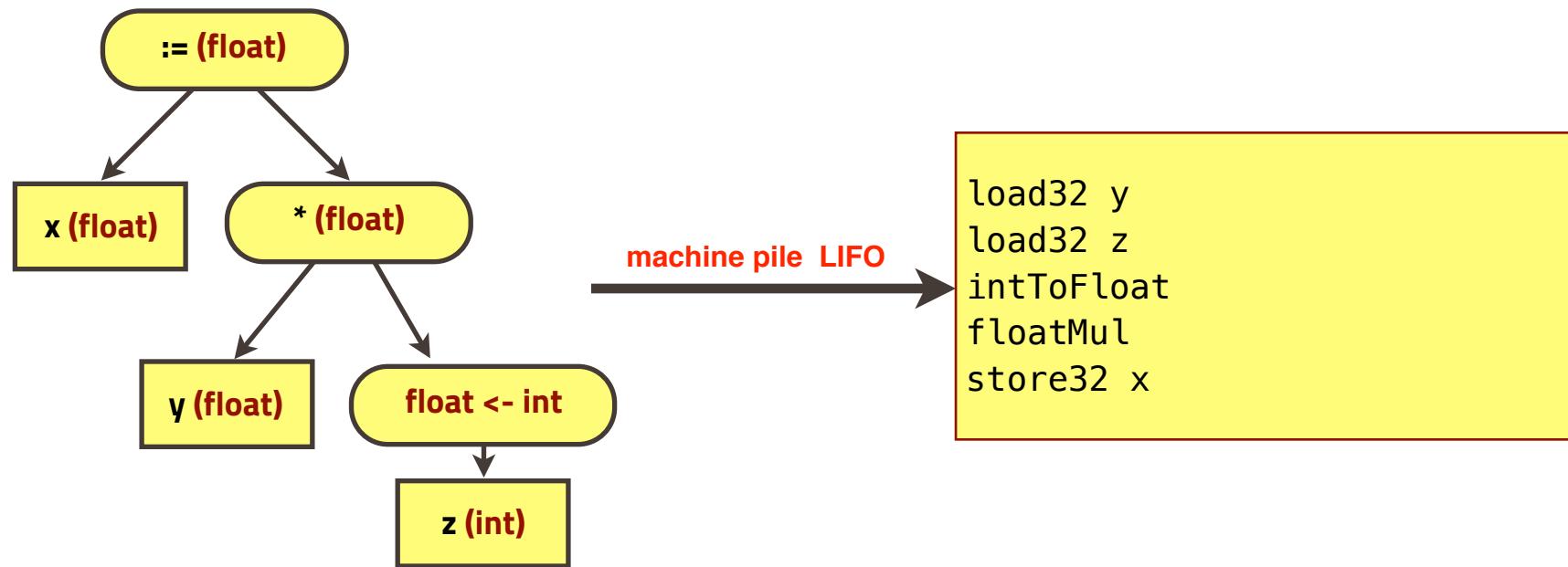
La génération de code inclut généralement des techniques d'optimisation :

- en taille de code engendré ;
- en temps d'exécution.

La génération de code peut aussi inclure des informations pour faciliter le débogage.

La génération de code (2/2)

Exemple de code engendré pour une machine à pile :



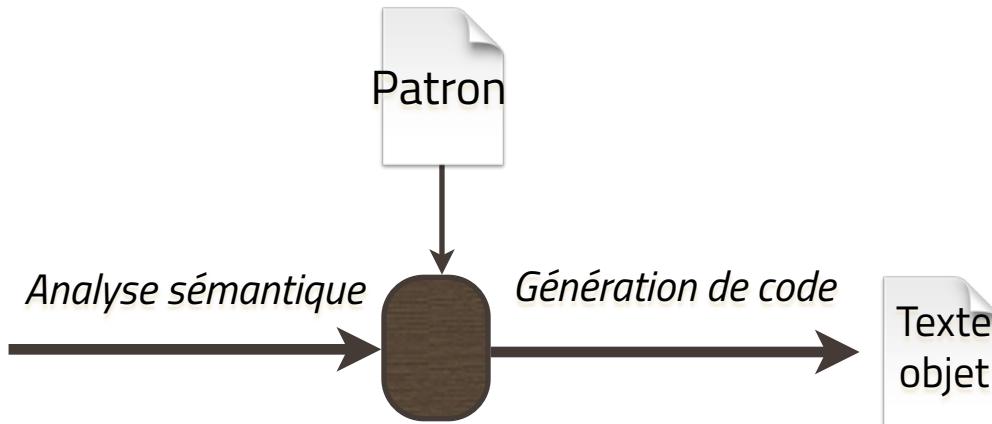
Génération de code par patron (« template »)

C'est un type particulier de générateur de code, qui engendre du texte source d'un format particulier.

Exemple pratique : comment engendrer une classe C++ ? Solution simple : par des appels à *printf* (ou autres...). Mais comment obtenir ces *printf* ? À la main ? Autre solution : utiliser en entrée un texte contenant des annotations décrivant les substitutions à effectuer pour obtenir le texte objet.

Exemples de langage de *template* :

- VTL (<http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html>) ;
- T4 (<http://msdn.microsoft.com/en-us/library/bb126445.aspx>).



Génération de code par patron : exemple pratique

Voici le texte objet que l'on voudrait obtenir.

```
class maClass {  
    public : maClass (void) ;  
    public : virtual ~ maClass (void) ;  
} ;
```

Voici un patron décrivant ce texte objet : il suffit de substituer \$CLASS\$ par le nom réel de la classe.

```
class $CLASS$ {  
    public : $CLASS$ (void) ;  
    public : virtual ~ $CLASS$ (void) ;  
} ;
```

Mais on peut imaginer une exigence plus importante : engendrer une liste d'attributs.

Pour analyser ce patron, il faut écrire un véritable analyseur lexical, syntaxique et sémantique.

```
class $CLASS$ {  
$for LISTE_ATTRIBUTS do$  
    protected : $TYPE$ $NOM$ ;  
$end$  
    public : $CLASS$ (void) ;  
    public : virtual ~ $CLASS$ (void) ;  
} ;
```

Application Binary Interface (ABI)

Application Binary Interface (http://en.wikipedia.org/wiki/Application_binary_interface) : ensemble des règles qui assurent l'interopérabilité des différentes chaînes de développement.

Plus précisément, pour chaque langage :

- la représentation interne des types de données,
- leur alignement, **données en mémoire**
- les conventions d'appel de fonctions, de routines,
- dans un langage objet, comment la liaison dynamique est implémentée, comment l'objet courant est représenté,
- ...

Pour les générateurs de code :

- le format des codes objet ;
- le format des bibliothèques ;

Ces règles sont primordiales pour assurer (entre autres) :

- l'interface entre l'assembleur et les autres langages ;
- l'appel des fonctions systèmes.

ABI : exemple de l'architecture ARM (1/3)

Une dizaine de documents, dont ceux-ci :

- *Procedure Call Standard for the ARM® Architecture* : http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf ;
- *C++ ABI for the ARM® Architecture* : http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041c/IHI0041C_cppabi.pdf

...

ABI : exemple de l'architecture ARM (2/3)

Les types fondamentaux de l'architecture ARM

Type Class	Machine Type	Byte size	Byte alignment	Note
Integral	Unsigned byte	1	1	Character
	Signed byte	1	1	
	Unsigned half-word	2	2	
	Signed half-word	2	2	
	Unsigned word	4	4	
	Signed word	4	4	
	Unsigned double-word	8	8	
	Signed double-word	8	8	
Floating Point	Half precision	2	2	See §4.1.1, <i>Half-precision Floating Point</i> .
	Single precision (IEEE 754)	4	4	The encoding of floating point numbers is described in [ARM ARM] chapter C2, <i>VFP Programmer's Model</i> , §2.1.1 <i>Single-precision format</i> , and §2.1.2 <i>Double-precision format</i> .
	Double precision (IEEE 754)	8	8	
Containerized vector	64-bit vector	8	8	See §4.1.2, <i>Containerized Vectors</i> .
	128-bit vector	16	8	
Pointer	Data pointer	4	4	Pointer arithmetic should be unsigned. Bit 0 of a code pointer indicates the target instruction set type (0 ARM, 1 Thumb).
	Code pointer	4	4	

ABI : exemple de l'architecture ARM (3/3)

Utilisation des registres par une procédure

Tous les registres sont 32 bits.

Lors d'un appel de procédure, le premier argument est rangé dans r0 par l'appelant, le deuxième dans r1, ... Si il y a plus de 4 arguments, les suivants sont placés dans la pile.

Les registres r4 à r8, r10 et r11 doivent être préservés par l'exécution d'une procédure. Si cette procédure est une fonction, le résultat est renvoyé dans r0.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

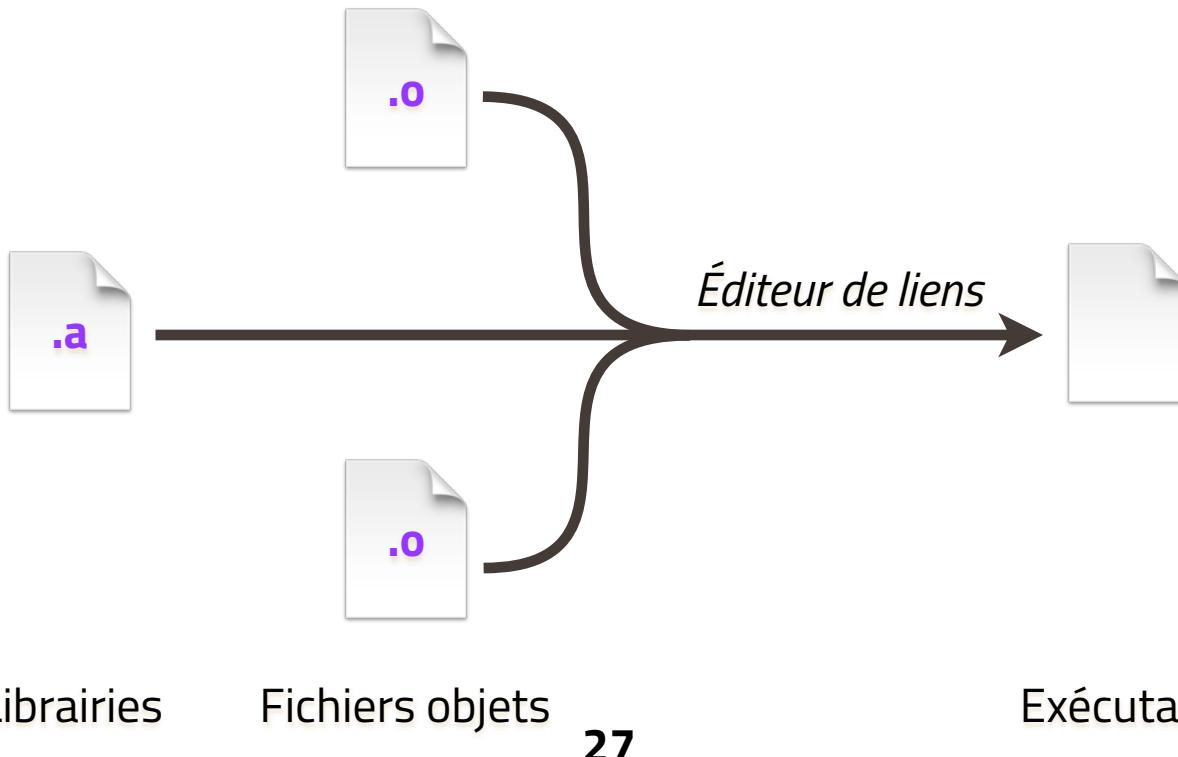
Rôle de l'éditeur de liens

Éditeur de liens : construire le code exécutable en

- rassemblant les différents codes objets ;
- résolvant les références ;
- important le code utile des librairies.

Besoins actuels :

- suppression des doublons issus de la compilation des templates C++ ;
- construction des tables des méthodes virtuelles ;
- optimisation inter-fichiers (*LTO, Link Time Optimization*).



Utilisation de GCC et binutils

GCC (GNU Compiler Collection) et binutils

GCC est une collection de compilateurs C, C++, Fortan, Java, ... et de librairies (`libc` pour le C, `libstdc++` pour le C++, ...).

GCC inclut des *backend* pour de nombreux processeurs, ce qui a fait sa popularité. Cross compiler GCC est *relativement simple*.

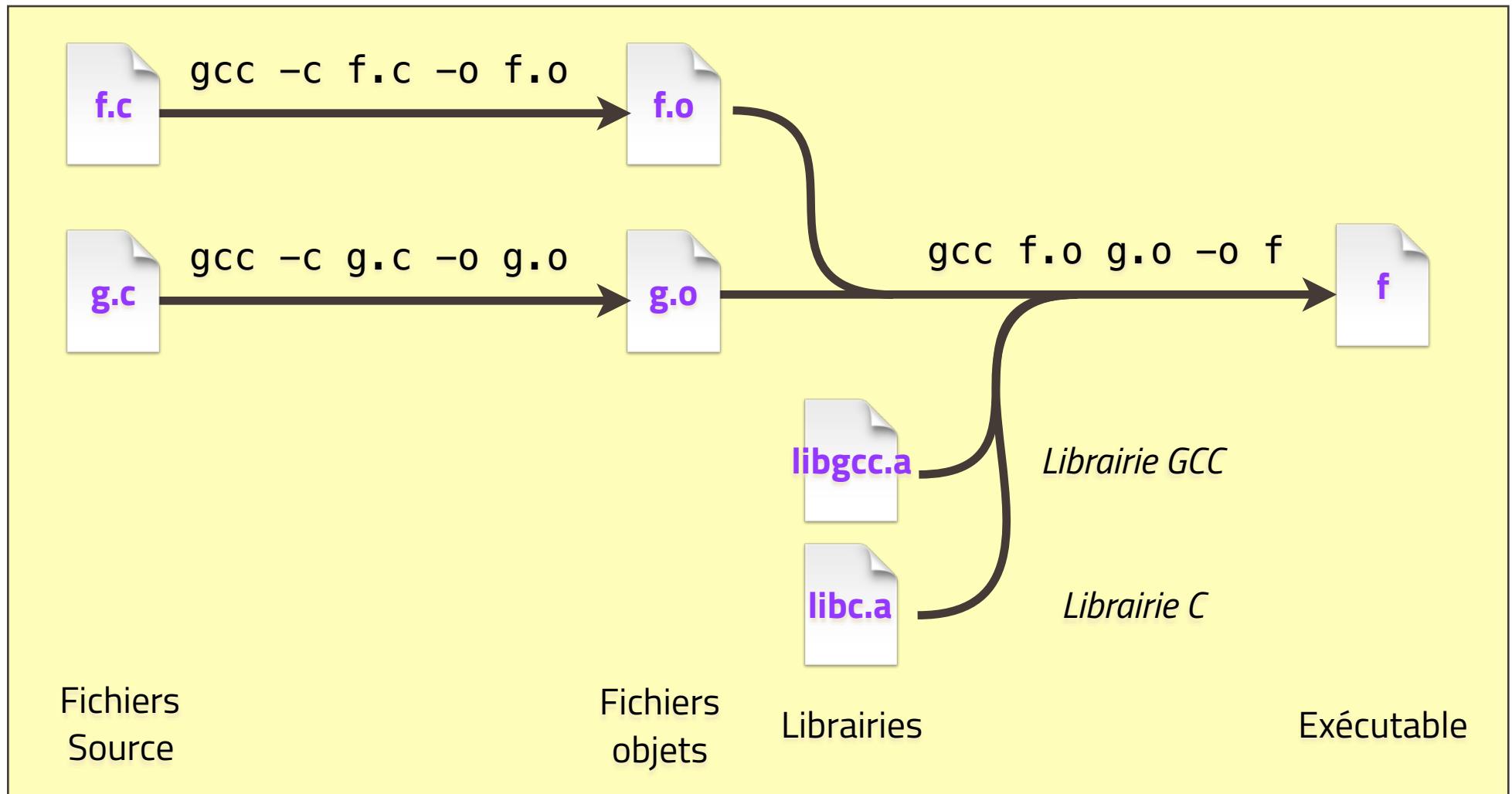
binutils est un ensemble d'utilitaires *backend* pour une chaîne de compilation, dont les deux principaux sont : **as**, l'assembleur, et **ld**, l'éditeur de liens.

Liens :

<https://gcc.gnu.org>

<https://www.gnu.org/software/binutils/>

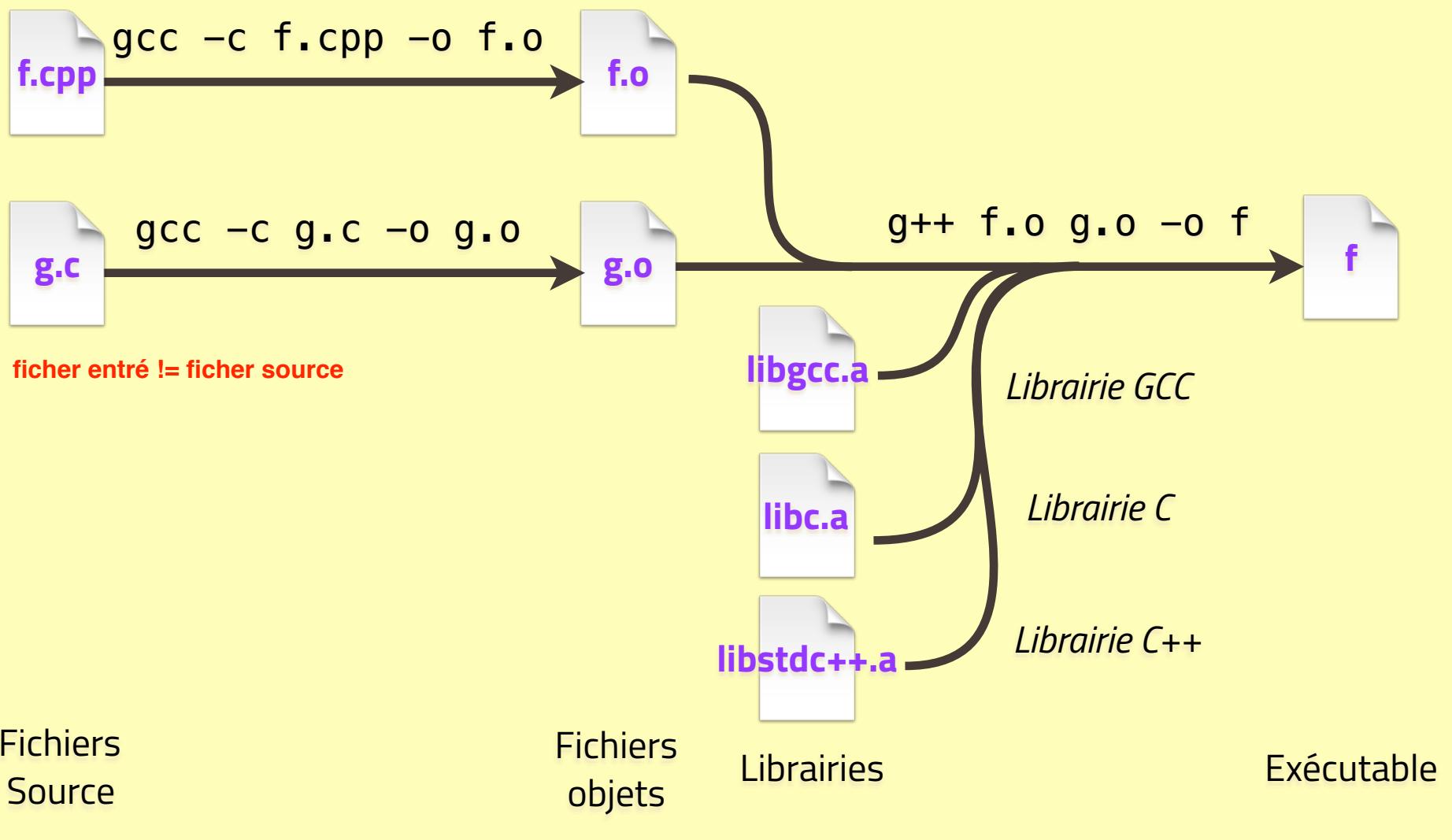
La chaîne de compilation C avec GCC



Principales options :

- `-c` : « *compile only* », n'effectue pas l'édition de liens ;
- `-o` : « *output* », désigne le fichier de sortie.

La chaîne de compilation C++ avec GCC



Ce que fait l'utilitaire gcc

L'utilitaire `gcc` est un utilitaire « chapeau » qui distribue en fait les opérations à réaliser en appelant les utilitaires appropriés :

- `cpp` : le pré-processeur ; **transformer les sources ensemble**
- `cc1` : le compilateur C (accepte un source sans aucune directive `#xxx`) ;
- `cc1plus` : le compilateur C++ (accepte un source sans aucune directive `#xxx`) ;
- `as` : l'assembleur ;
- `ld` : l'éditeur de liens.

L'intérêt d'appeler `gcc` est qu'il appelle automatiquement, avec les options appropriées, les utilitaires nécessaires : cela simplifie l'utilisation de GCC.

Note : `as` et `ld` ne font pas partie de `gcc`, mais de `binutils`. Installer une chaîne complète de compilation doit comprendre `gcc` et `binutils`.

Les librairies

Plusieurs librairies entrent en jeu :

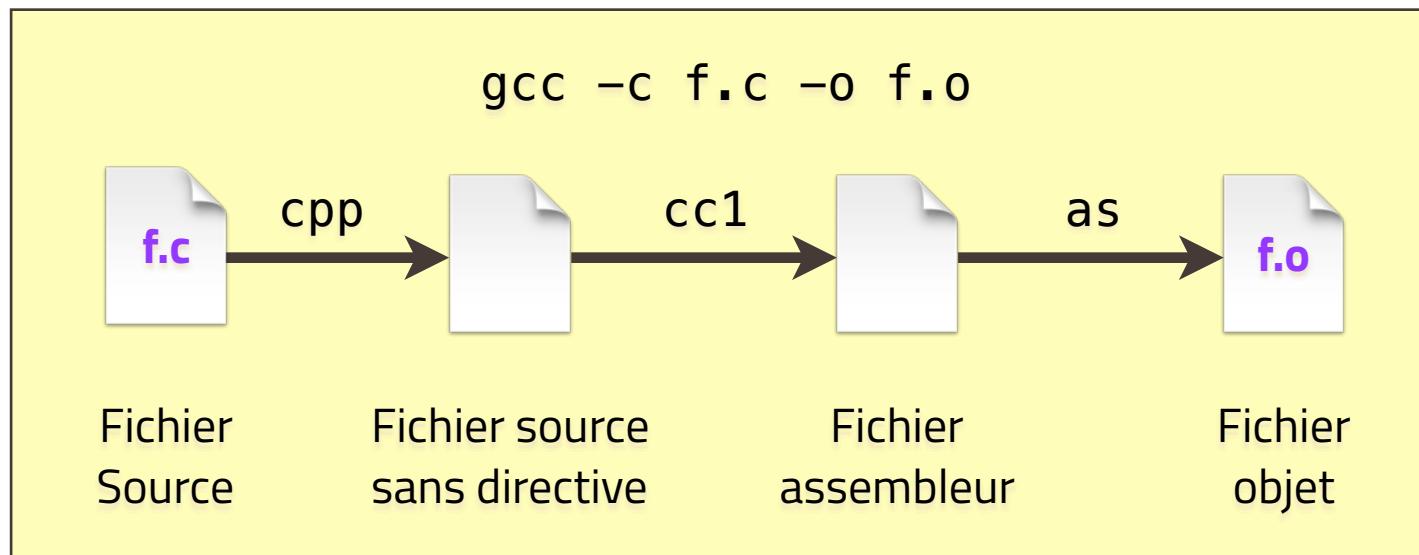
- **libgcc** : librairie de bas niveau, nécessaire quelque soit le langage ; contient :
 - * la mise à zéro d'une zone mémoire, copie de zones mémoire ;
 - * si nécessaire, émulation des calculs en flottants ;
 - * opérations arithmétiques ;
 - * l'interface avec le système d'exploitation ;
 - * ...
- **libc** : librairie liée au langage C ;
- **libc++** : librairie liée au langage C++.

L'intérêt d'appeler `gcc` est qu'il appelle automatiquement, avec les options appropriées, les utilitaires nécessaires : cela simplifie l'utilisation de GCC.

Note : la librairie `libc` ne fait pas partie de GCC. Elle est apportée par `newlib` ou `glibc`. Par contre, la librairie C++ fait partie de GCC.

Détail d'utilisation de gcc (1/3)

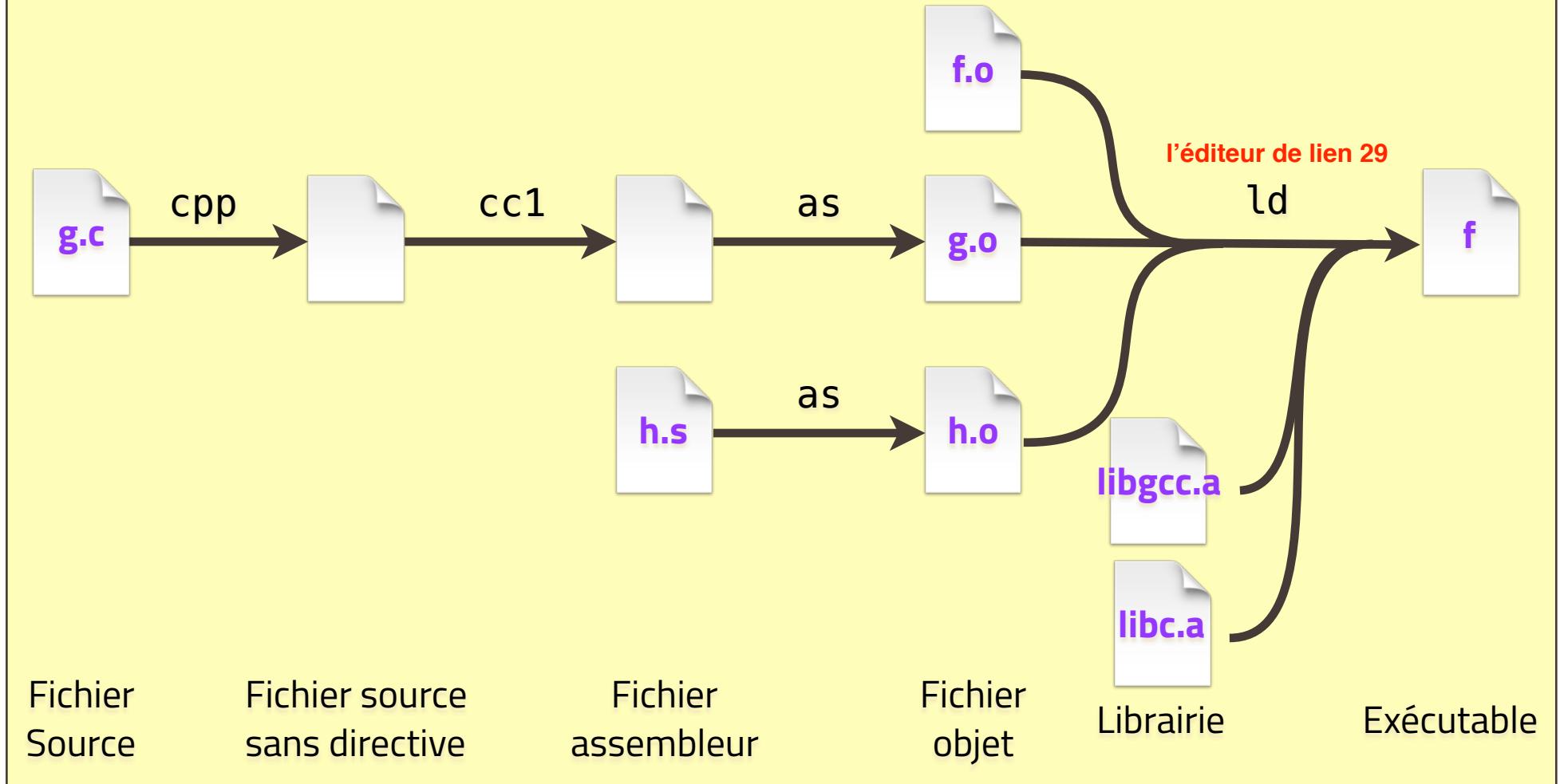
Les intermédiaires sont par défaut des fichiers ou des *pipes* (option `-pipe`).



Détail d'utilisation de gcc (2/3)

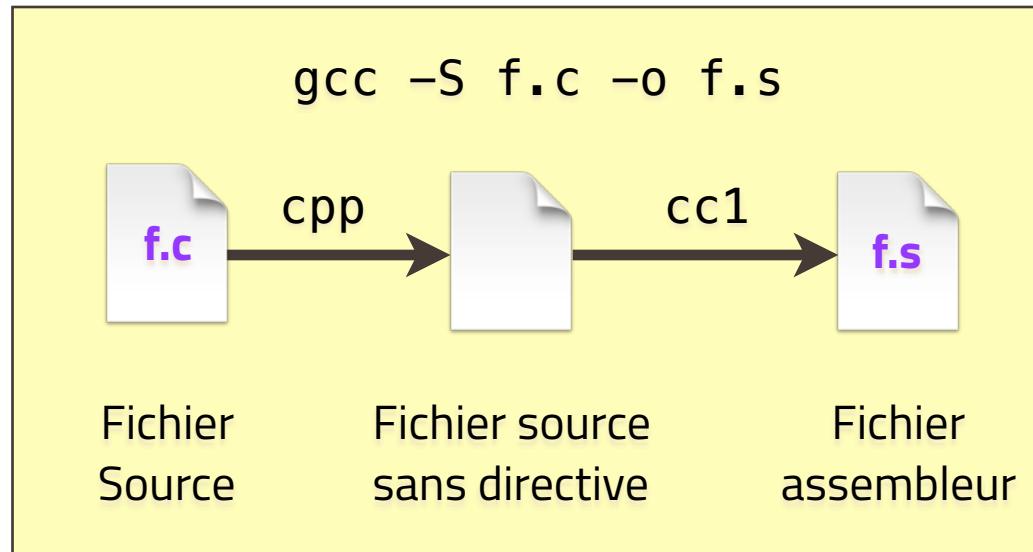
On peut combiner compilation, assemblage et édition de liens

```
gcc f.o g.c h.s -o f
```



Détail d'utilisation de gcc (3/3)

Option –S : produire un fichier assembleur



Intérêt : examiner le code assembleur engendré par la compilation.

Cross compilateurs

Cross compilateur, exemple de GCC (1/3)

« *Sur mon Macintosh, je construis un compilateur pour processeur ARM qui fonctionnera sous Linux.* »

Bibliographie : http://en.wikipedia.org/wiki/Cross_compiler

Quand on recompile GCC, trois paramètres précisent le compilateur engendré ([3], pages 233 et 234) :

- **--build** : le système sur lequel le compilateur est construit (le plus souvent implicite) ;
- **--host** : le système sur lequel tournera le compilateur engendré ;
- **--target** : le système cible du compilateur.

Cross compilateur, exemple de GCC (2/3)

Illustration ([3], pages 233 et 234) :

Table 11-1. Compiler Types				
Build	Host	Target	Compiler Type	Result
x86	x86	x86	Native	Built on an x86 to run on an x86 to generate binaries for an x86
SH	SH	ARM	Cross	Built on a SuperH to run on a SuperH to generate binaries for an ARM
x86	MIPS	x86	Crossback	Built on an x86 to run on a MIPS to generate binaries for an x86
PPC	SPARC	SPARC	Crossed native	Built on a PPC to run on a SPARC to generate code for a SPARC
ARM	SH	MIPS	Canadian	Built on an ARM to run on a SuperH to generate code for a MIPS

En fait, la désignation de ces paramètres de configuration des systèmes qui est illustrée ici est imprécise (voir page suivante).

Cross compilateur, exemple de GCC (3/3)

Pour GCC, un système est spécifié par un triplet :

- l'architecture ;
- le vendeur ;
- le système d'exploitation.

Par exemple :

- powerpc-apple-darwin ;
- i386-pc-linux ;
- x86_64-pc-linux ;
- i386-pc-mingw32.

Mais cross compiler GCC n'est pas toujours simple !

Exemple de code engendré (processeur ARM)

Programme C d'exemple :

```
unsigned somme (const unsigned inCount) {
    unsigned result = 0 ;
    unsigned i ;
    for (i=1 ; i<=inCount ; i++) {
        result += i ;
    }
    return result ;
}
```

Exemple de code assembleur engendré

Commande : **arm-elf-gcc -O2 -fomit-frame-pointer -S somme.c -o somme.s**

```
.file    "somme.c"
.text
.align   2
.global  somme
.type    somme, %function
somme:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 0, uses_anonymous_args = 0
@ link register save eliminated.
cmp r0, #0
@ lr needed for prologue
moveq   r2, r0
beq .L4
mov r2, #0
                                mov r3, #1
.L5:
add r2, r2, r3
add r3, r3, #1
cmp r0, r3
bcs .L5
.L4:
mov r0, r2
bx lr
.size   somme, .-somme
compilateur: .ident  "GCC: (GNU) 4.2.0"
```

Exemple de code machine engendré

Commandes :

```
arm-elf-gcc -O2 -fomit-frame-pointer -c somme.c -o somme.o  
arm-elf-objdump -d somme.o
```

```
somme.o:      file format elf32-littlearm

Disassembly of section .text:

00000000 <somme>:
 0:   e3500000      cmp r0, #0    ; 0x0
 4:   01a02000      moveq r2, r0
 8:   0a000005      beq 24 <somme+0x24>
 c:   e3a02000      mov r2, #0    ; 0x0
10:   e3a03001      mov r3, #1    ; 0x1
14:   e0822003      add r2, r2, r3
18:   e2833001      add r3, r3, #1    ; 0x1
1c:   e1500003      cmp r0, r3
20:   2afffffb      bcs 14 <somme+0x14>
24:   e1a00002      mov r0, r2
28:   e12fff1e      bx lr
```

LLVM - CLANG

LLVM — clang

Initiative universitaire sponsorisée par Apple pour offrir un remplacement de GCC.

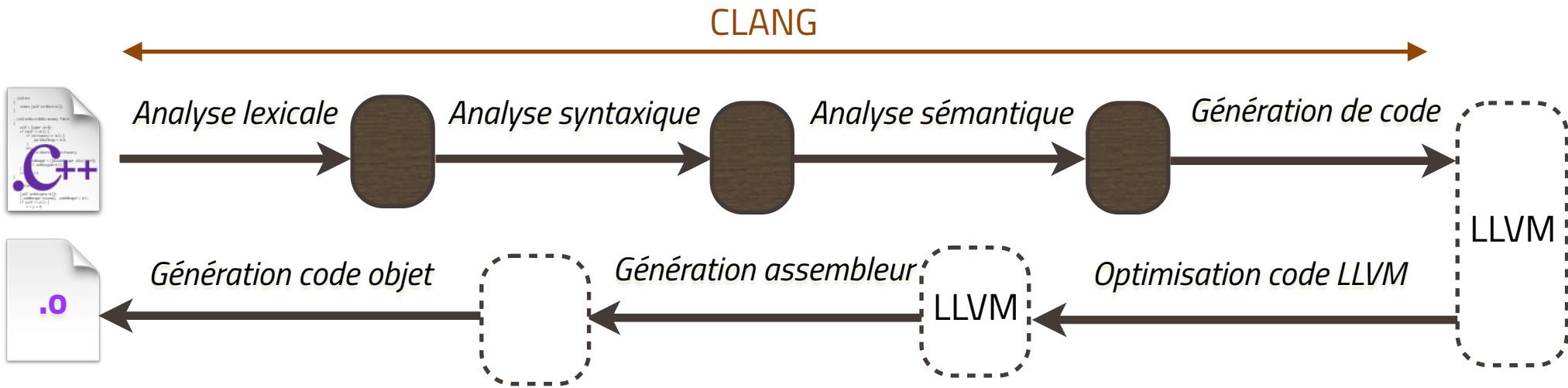
LLVM : *Low Level Virtual Machine* (<http://llvm.org/>)

clang : front-end des langages C, C++, Objective C pour LLVM (<http://clang.llvm.org/>)

Low Level Virtual Machine (LLVM) is:

1. *A compilation strategy designed to enable effective program optimization across the entire lifetime of a program. LLVM supports effective optimization at compile time, link-time (particularly interprocedural), run-time and offline (i.e., after software is installed), [...].*
2. *A virtual instruction set - LLVM is a low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and dataflow (SSA) information about operands. [...].*
3. *A compiler infrastructure - LLVM is also a collection of source code that implements the language and compilation strategy. [...]*
4. [...]

LLVM — clang



L'articulation entre *front-end* et *back-end* est le code intermédiaire LLVM.

clang correspond à *GCC* en se limitant à la génération de code intermédiaire.

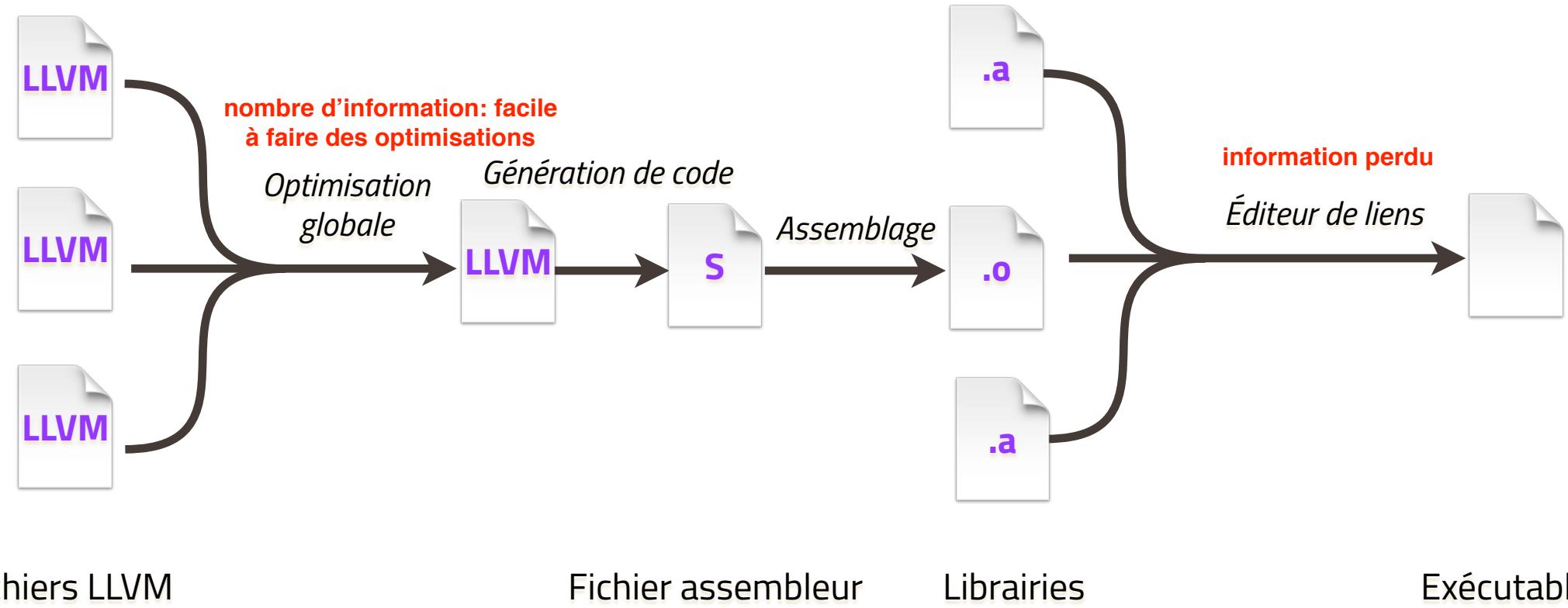
LLVM regroupe la transformation du code intermédiaire en code machine, et les outils sur ces codes.

Obtenir LLVM — clang : <http://llvm.org/releases/>

Compiler LLVM — clang : http://clang.llvm.org/get_started.html

Tutoriel d'écriture d'un front-end pour LLVM : Kaleidoscope (<http://llvm.org/docs/tutorial/>).

Link Time Optimization avec LLVM



Utiliser LLVM — clang

Voir à la page : http://clang.llvm.org/get_started.html

Compiler et lier :

```
clang file.c -O3 -o file
```

Uniquement l'analyse syntaxique :

```
clang file.c -fsyntax-only
```

Imprimer le code LLVM engendré :

```
clang file.c -S -emit-llvm -o -
```

Imprimer le code LLVM engendré, optimisé :

```
clang file.c -O3 -S -emit-llvm -o -
```

Imprimer le code assembleur engendré, optimisé :

```
clang file.c -O3 -S -o -
```

Exemple clang — LLVM

Code C :

```
unsigned somme (const unsigned inCount)
{
    unsigned result = 0 ;
    unsigned i ;
    for (i=1 ; i<=inCount ; i++) {
        result += i ;
    }
    return result ;
}
```

Code LLVM engendré

3 blocks

```
define i32 @somme(i32 %inCount) #0 {
    %1 = icmp eq i32 %inCount, 0
    br i1 %1, label %.crit_edge, label %.lr.ph

    .lr.ph: ; preds = %0, %.lr.ph
    %i.02 = phi i32 [ %3, %.lr.ph ], [ 1, %0 ]
    %result.01 = phi i32 [ %2, %.lr.ph ], [ 0, %0 ]
    %2 = add i32 %i.02, %result.01
    %3 = add i32 %i.02, 1
    %4 = icmp ugt i32 %3, %inCount
    br i1 %4, label %.crit_edge, label %.lr.ph

    .crit_edge: ; preds = %.lr.ph, %0
    %result.0.lcssa = phi i32 [ 0, %0 ], [ %2, %.lr.ph ]
    ret i32 %result.0.lcssa
}
```

Points particuliers de ce langage intermédiaire :

- langage de blocs (pas d'exécution structurée, que des branchements conditionnels ou inconditionnels) ;
- pas de variable, que des constantes ;
- instruction *phi*.

Compilateurs utilisés en TP

Lex — Yacc

Lex ([http://en.wikipedia.org/wiki/Lex_\(software\)](http://en.wikipedia.org/wiki/Lex_(software))) est un générateur d'analyseur lexical qui produit un texte C.

Il fonctionne conjointement avec **YACC** (<http://en.wikipedia.org/wiki/Yacc>), générateur d'analyseur syntaxique. YACC accepte les grammaires LALR (http://en.wikipedia.org/wiki/LALR_parser).

Mini manuel d'utilisation : <http://www.linux-france.org/article/devl/lexyacc/>

GALGAS

GALGAS (<http://galgas.rts-software.org/>) est un générateur de compilateur développé à l'IRCCyN.

C'est un ensemble de langages dédiés (DSL) pour :

- décrire l'analyseur lexical ;
- l'analyseur syntaxique,
- la grammaire ;
- l'analyse sémantique ;
- la génération de code.

Construire une chaîne de développement

Construire une chaîne de développement (pour système embarqué)

Construire une chaîne pour système embarqué peut, dans certains cas, présenter un intérêt :

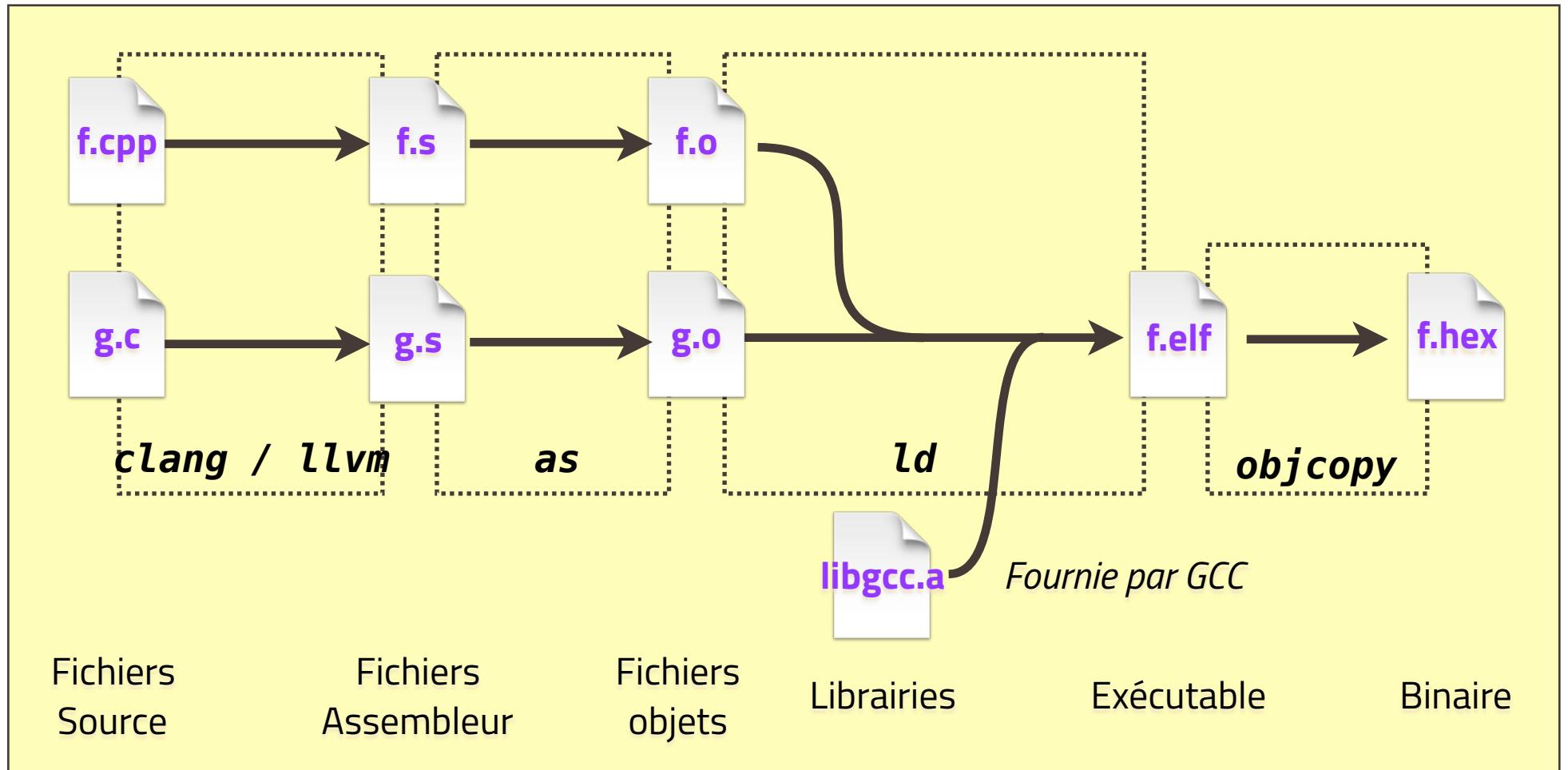
- compléter une chaîne existante (par exemple LLVM / CLANG ne contient pas d'assembleur ni d'éditeur de liens) ;
- avoir un cross-compilateur (sur Mac, je compile pour obtenir un exécutable Windows) ;
- assurer l'identité de développement entre entre plateformes en fournissant les mêmes outils.

La situation change régulièrement : ainsi, actuellement ARM fournit les binaires pour Linux, Mac OS, Windows d'un portage de GCC pour processeurs ARM :

<https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm>

Construire sa propre chaîne GCC pour ARM n'a plus guère de raison d'être.

Construire une chaîne de développement (pour système embarqué)



CLANG / LLVM ne contient ni assembleur ni éditeur de liens, on utilise les utilitaires `as`, `ld` et `objcopy` issus de *binutils*.

Construire une chaîne de développement (pour système embarqué)

Les commandes à exécuter peuvent être difficiles à mettre au point :

- cross-compiler GCC 9 pour ~~Windows~~ sur Mac échoue sur de nombreux systèmes Mac OS... mais réussit sur ~~Lion~~ (~~10.6~~) ;
~~Snow Leopard~~
- parfois (souvent) de nombreux paramètres à ajuster ;
- cela peut s'avérer si difficile que l'on abandonne...

Pour simplifier la mise au point :

- écrire un script que l'on lance simplement, et qui effectue le travail par étapes (un *shell script*, ou mieux un script Python) ;
- ainsi, si une étape échoue, on peut la relancer sans refaire les précédentes qui ont réussi ;
- télécharger les archives au début de l'exécution du script
 - ▶ évidemment pas à chaque fois, les mémoriser dans un répertoire ;
 - ▶ utiliser un outil téléchargement externe à Python, par exemple `curl` (on confie au système le soin de vérifier les certificats).

Un exemple de script

Construit sur Mac OS une chaîne de développement pour la carte Teensy 3.6 (processeur ARM Cortex-M4) et carte LPC2294 (processeur ARM7TDMI) :

- construit binutils ;
- construit CLANG / LLVM
- pour cette dernière opération, on a besoin de :
 - xy : les archives de LLVM / CLANG sont dans ce format, et Mac OS n'a pas d'utilitaire de désarchivage de ce format ;
 - cmake : organise la construction de clang / llvm ;
 - il faut donc d'abord télécharger et compiler ces outils ;
- pour télécharger le code dans la cible on a besoin d'un autre utilitaire :
 - teensy-cli-loader (on télécharge et compile le source) ;
 - openocd (interface avec une sonde JTAG) ;
 - openocd a besoin de libusb (prise en charge de l'USB) ;
- ...

Extrait de l'exécution de ce script

```
+ cp /Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/archives-for-cross-compilation/binutils-2.33.1.tar.bz2 binutils-2.33.1.tar.bz2
+ bunzip2 binutils-2.33.1.tar.bz2
+ tar xf binutils-2.33.1.tar
+ rm binutils-2.33.1.tar
+ mkdir build-binutils
+ chdir /Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/build-binutils
+ ../binutils-2.33.1/configure --help
-----
+ ../binutils-2.33.1/configure --target=arm-eabi --prefix=/Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/omnibus-Darwin-i386-llvm-9.0.0-binutils-2.33.1-openocd-0.10.0-libusb-1.0.23 --with-sysroot=/Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/omnibus-Darwin-i386-llvm-9.0.0-binutils-2.33.1-openocd-0.10.0-libusb-1.0.23/arm-eabi --enable-interwork --enable-multilib --disable-werror --disable-libstdcxx 'CFLAGS=-O2 -fomit-frame-pointer' 'CXXFLAGS=-O2 -fomit-frame-pointer'
-----
+ make all -j13
-----
+ make install
-----
+ chdir /Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd
+ remove '/Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/build-binutils' directory
+ remove '/Volumes/dev-svn/omnibus-dev/arm-clang-llvm-binutils-openocd/binutils-2.33.1' directory
```