Introduction au projet COCass

Amélie Ledein

Inspirée du cours de **Ivan Augé** et **Guillaume Burel**

Et le sujet du projet est...

Programmer en OCaml un compilateur du langage C−− vers de l'assembleur Intel x86.

Pour vous aider, vous avez des fichiers sur ma page.
Et n'hésitez pas à poser des questions !

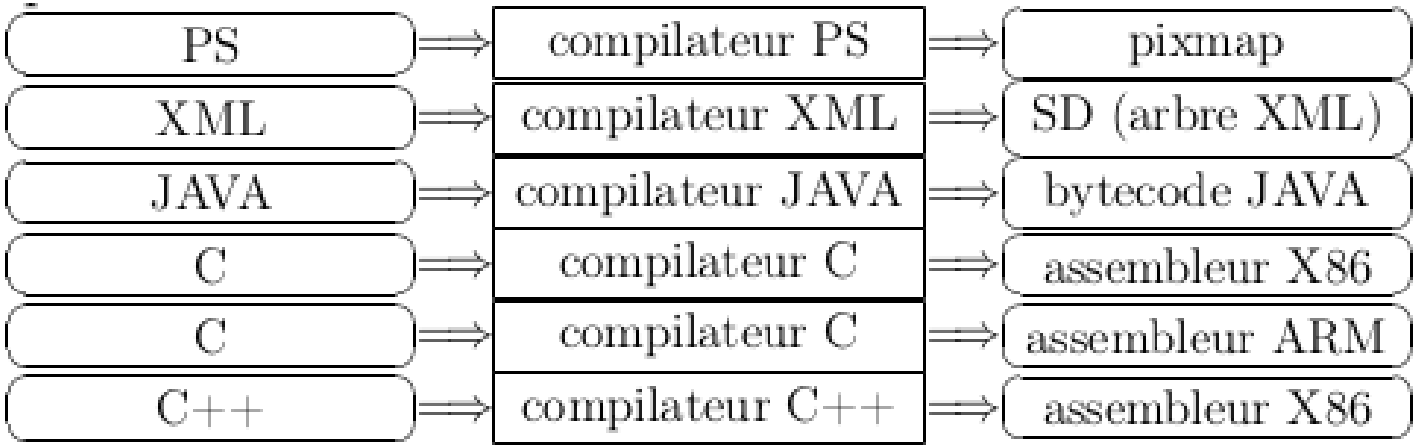
Vous avez dit un compilateur ?

Un **compilateur** est un programme qui lit un flux d'entrée structuré par une grammaire et qui produit une sortie.

L'entrée et la sortie sont des langages de programmation.

Nous pouvons donc aussi dire qu'un **compilateur** est un exécutable qui traduit un langage de haut niveau vers un langage de plus bas niveau.

Exemples :



Plus précisément

Un **compilateur** ne se contente pas de traduire un langage dans un autre, il est capable de :

- signaler des erreurs de syntaxe,
- signaler des erreurs de sémantique,
- faire des optimisations (vitesse d'exécution, taille du code, utilisation de la mémoire, etc.).

Comment procède un compilateur ?

Les **étapes de compilation** sont :

$$
\begin{array}{rcl}
\text{source} & \Longrightarrow & \text{analyse lexicale} \Longrightarrow \{\ (t_0,v_0)\ (t_1,v_1),\ \ldots\ \} \\
\{\ (t_0,v_0)\ (t_1,v_1),\ \ldots\ \} & \Longrightarrow & \text{analyse grammaticale} \Longrightarrow \text{AST}^* \\
\text{AST} & \Longrightarrow & \text{analyse sémantique} \Longrightarrow \text{SD} \\
\text{SD} & \Longrightarrow & \text{génération} \Longrightarrow \text{sortie}
\end{array}
$$

Vocabulaire :

- **Analyse lexicale**, ou **lexing** en anglais
- t_i = un jeton (ou **token** en anglais)
- v_i = une **valeur**
- **Analyse grammaticale**, ou **parsing** en anglais
- **AST** = Abstract syntax tree, i.e. **arbre de syntaxe abstraite**
- **Analyse syntaxique** = Analyse lexicale + grammaticale

ATTENTION : Certains ouvrages parlent d'analyse syntaxique au lieu d'analyse grammaticale.

Analyse lexicale & grammaticale

- Outils historiques : Lex & Yacc
- Note : Lex et Yacc sont aussi des compilateurs : ils traduisent des expressions régulières et des grammaires hors-contexte vers du code C. (Cf. l'acronyme de Yacc :Yet Another Compiler Compiler).
- Outils OCaml : ocamllex, ocamlyacc
- Dans le cadre de ce projet :
  - Le lexeur se trouve dans le fichier `clex.mll`.
  - Le parseur se trouve dans le fichier `cparse.mly`, et a besoin du fichier `cAST.ml`.
  - Vous n'avez donc pas à coder l'analyse syntaxique !

Analyse lexicale

Objectif : Reconnaître une séquence de mots appartenant à un langage défini à l'aide d'une expression régulière.
Nous utilisons pour cela des techniques utilisant des automates finis.

- Extrait du fichier `cparse.mly` :

```
%token CHAR DOUBLE
%token ENUM
%token CASE ELSE FOR BREAK
```

- Extrait du fichier `clex.mll` :

```
rule ctoken = parse
| "break" { count (Lexing.lexeme lexbuf); BREAK }
| "case" { count (Lexing.lexeme lexbuf); CASE }
| "char" { count (Lexing.lexeme lexbuf); CHAR }
| "double" { count (Lexing.lexeme lexbuf); DOUBLE }
| "else" { count (Lexing.lexeme lexbuf); ELSE }
| "enum" { count (Lexing.lexeme lexbuf); ENUM }
| "for" { count (Lexing.lexeme lexbuf); FOR }
```

Analyse sémantique

L'analyse sémantique étudie l'arbre de syntaxe abstraite produit par l'analyse syntaxique pour éliminer au maximum les programmes qui ne sont pas corrects du point de vue de la sémantique.

Exemples :

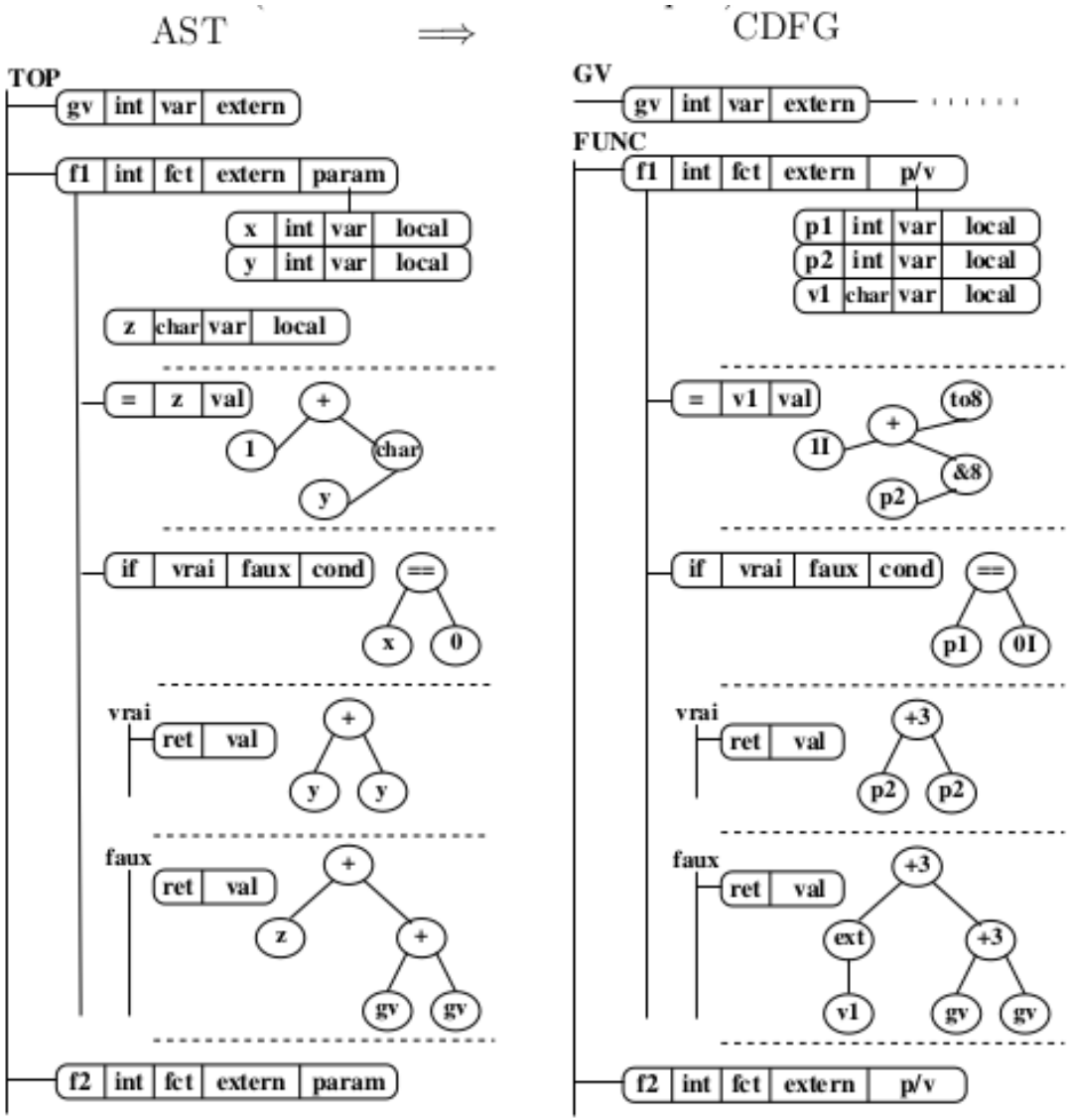
- Vérifier la portée des variables
- Vérifier le typage des expressions

  $\rightarrow$ Ce n'est pas votre objectif dans le cadre de ce projet.

  $\rightarrow$ Votre objectif est de générer du code assembleur à partir d'un AST.

Analyse sémantique - Exemple courant

Pour ce faire, l'analyse sémantique part d'un arbre syntaxique abstrait (AST) et génère un CDFG (Control Data Flow Graphe).



Séance 1

1. Récupérer les fichiers du projet qui sont sur ma page.
2. Faire `make`, et me dire si tout fonctionne pour vous.
3. Comprendre ce qui s'est passé quand vous avez fait `make`.
4. Modifier le fichier `compile.ml` pour commencer le projet.

Qu'est-ce que C−− ?

Souvenez-vous du cours 6, où vous avez découvert la syntaxe de IMP :

Commandes

c	$::=$	$x := e$	affectation
	$\mid$	`skip`	ne rien faire
	$\mid$	$c_1; c_2$	séquence
	$\mid$	`if` e `then` c_1 `else` c_2	conditionnelle
	$\mid$	`while` e `do` c	boucle while

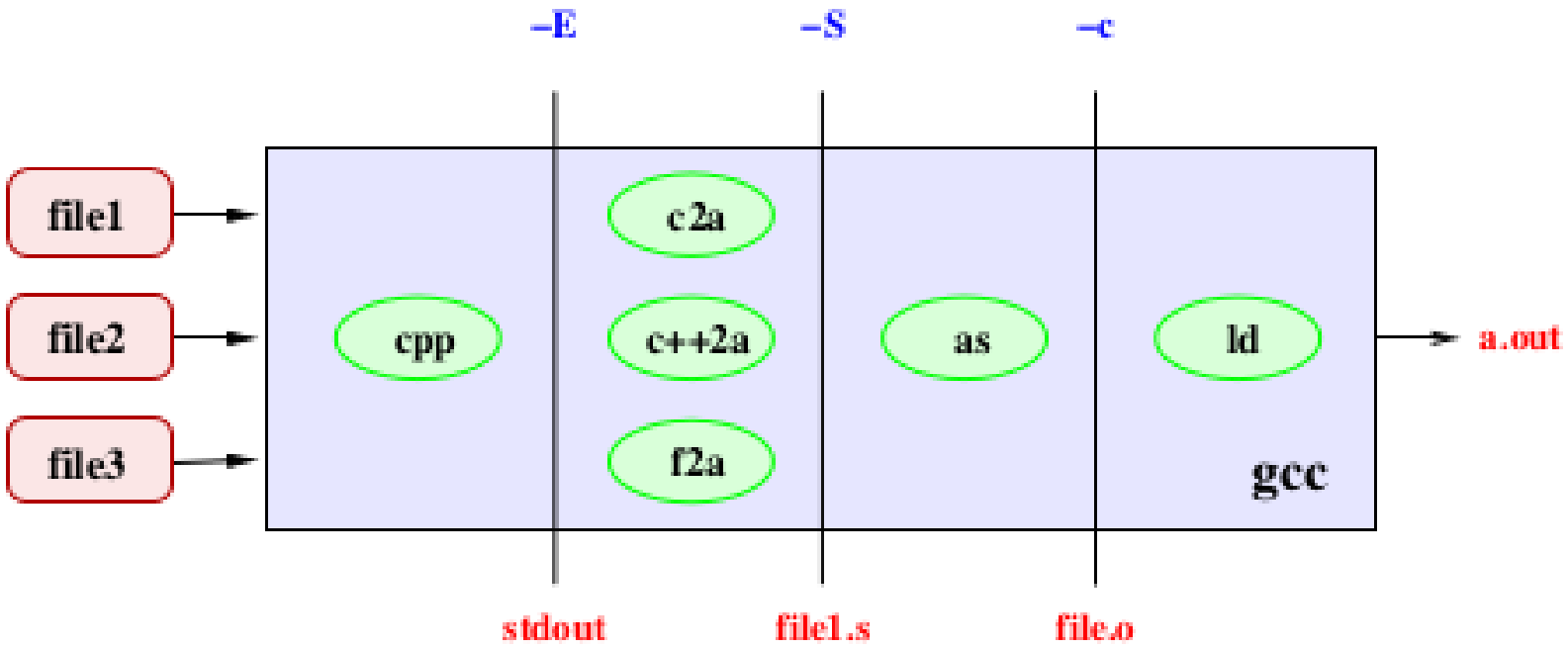
Expressions

e	$::=$	x	variables
	$\mid$	$\dot{n}$	constante entière ($n \in \mathbb{Z}$)
	$\mid$	$e \dot{+} e$	addition
	$\mid$	$\dot{-} e$	opposé

Exercice : Ecrire la syntaxe de C−− à l'aide des fichiers du projet.

Chaine de compilation

- Vous pouvez demander à gcc de s'arrêter dans la chaine de compilation en utilisant les options `-E`, `-S`, `-c` :



`gcc -S -fno-asynchronous-unwind-tables Exemples/cat.c` pour ne pas voir les optimisations que le compilateur fait.

- Pour voir ce que votre compilateur fait : `./mcc -S Exemples/cat.c`
- N'oubliez pas l'outil disponible ici `https://godbolt.org/` (Désactivez Output > "Intel asm syntax")

Séance 2

1. Récupérer les fichiers du projet qui sont sur ma page, car j'ai corrigé quelques imprécisions.
2. Ecrire la syntaxe de C−− en vous aidant des fichiers du projet.
3. Visualiser vos premiers AST, sur des exemples simples, à l'aide la fonction fournie `print_ast` (fichier `cprint.ml`).
   Note : Cette fonction s'utilise avec la commande `./mcc -A Exemples/cat.c` .
4. Comparer, sur des exemples simples, à quoi correspond le code assembleur à l'aide de l'outil disponible ici `https://godbolt.org/`.
5. Réussir l'objectif final sur les fichiers `ex0.c`, `ex1.c` et `ex2.c`, disponibles sur ma page.
6. Continuer le travail par incrémentation pour considérer le sous-ensemble de C−− qui correspond au langage arithmétique.

Votre mission, si vous l'acceptez ...

- Pour aujourd'hui :
  - Continuer le travail effectué jusqu'ici, par incrémentation.
  - N'hésitez pas si vous avez des difficultés.

Objectifs à long terme

- Passer d'un AST à un exécutable, i.e. compléter le fichier `compile.ml`, principalement.
- Faire une présentation de votre travail en janvier.