Introduction à la compilation
Cours 1 : Qu'est-ce qu'un compilateur?

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

Qu'est-ce que la compilation ?

- Vous avez tous déjà tapé :

```
% javac Fact.java
```

- Que se passe-t-il ensuite ?

Qu'est-ce qu'un compilateur ?

- Il se peut que votre programme soit rejeté :

```
Fact.java:3: incompatible types
found : boolean
required: int
      if (n == 0) return (true);
                                ^
1 error
```

- Il se peut aussi que votre programme soit accepté.
  Dans ce cas, vous pouvez exécuter le programme :

```
% java Fact
2004310016
```

Comment fonctionne un compilateur ?

- C'est le sujet de ce cours.
- Aujourd'hui, nous allons tenter de répondre à ces trois questions :
  1. Qu'attend-t-on du compilateur ?
  2. Quels sont les techniques d'implémentation utilisées par la compilation ?
  3. Pourquoi étudier la compilation ?

Qu'attend-t-on du compilateur ?

Qu'attend-t-on du compilateur ?

- Pour donner une spécification à un compilateur, il faut d'abord bien comprendre ce qu'est la *programmation*.
- Informellement, la programmation est l'art de résoudre des problèmes e□cacement, par le calcul.

Résoudre des problèmes ?

- Il existe de nombreux types de problèmes : certains sont simples à résoudre et d'autres sont plus di□ciles, certains ont une solution connue, d'autres pas, certains ont un énoncé très court tandis que d'autres nécessitent une description gigantesque.
- Dans tous les cas, cependant, un problème, bien posé, peut se spécifier à l'aide d'une formule logique de la forme :

$$\forall I, P(I) \Rightarrow \exists O, Q(I, O)$$

- où :
  - P est la précondition du problème : le domaine de ses entrées valides.
  - Q est sa postcondition : la relation attendue entre les entrées et les sorties.

Exemple de spécification

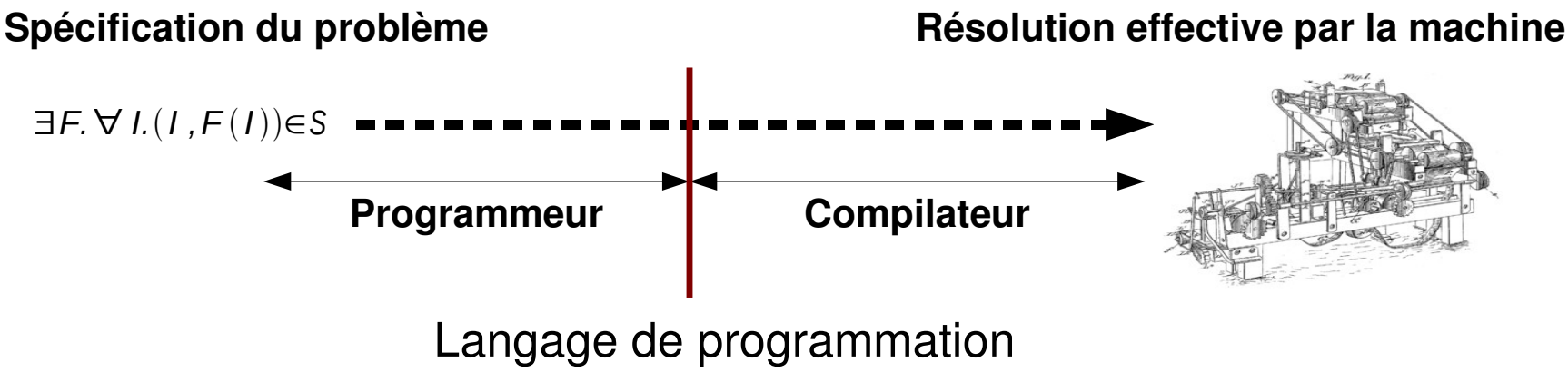
$$\forall I,\ Precondition(I) \implies \exists O,\ Postcondition(I, O)$$

- Un algorithme de tri :
  - *Entrée* : un tableau t.
  - *Précondition* : il existe une relation d'ordre sur les éléments du tableau.
  - *Sortie* : un tableau u.
  - *Postcondition* : u est trié et contient exactement les mêmes éléments que t.

Distance entre spécification et réalisation

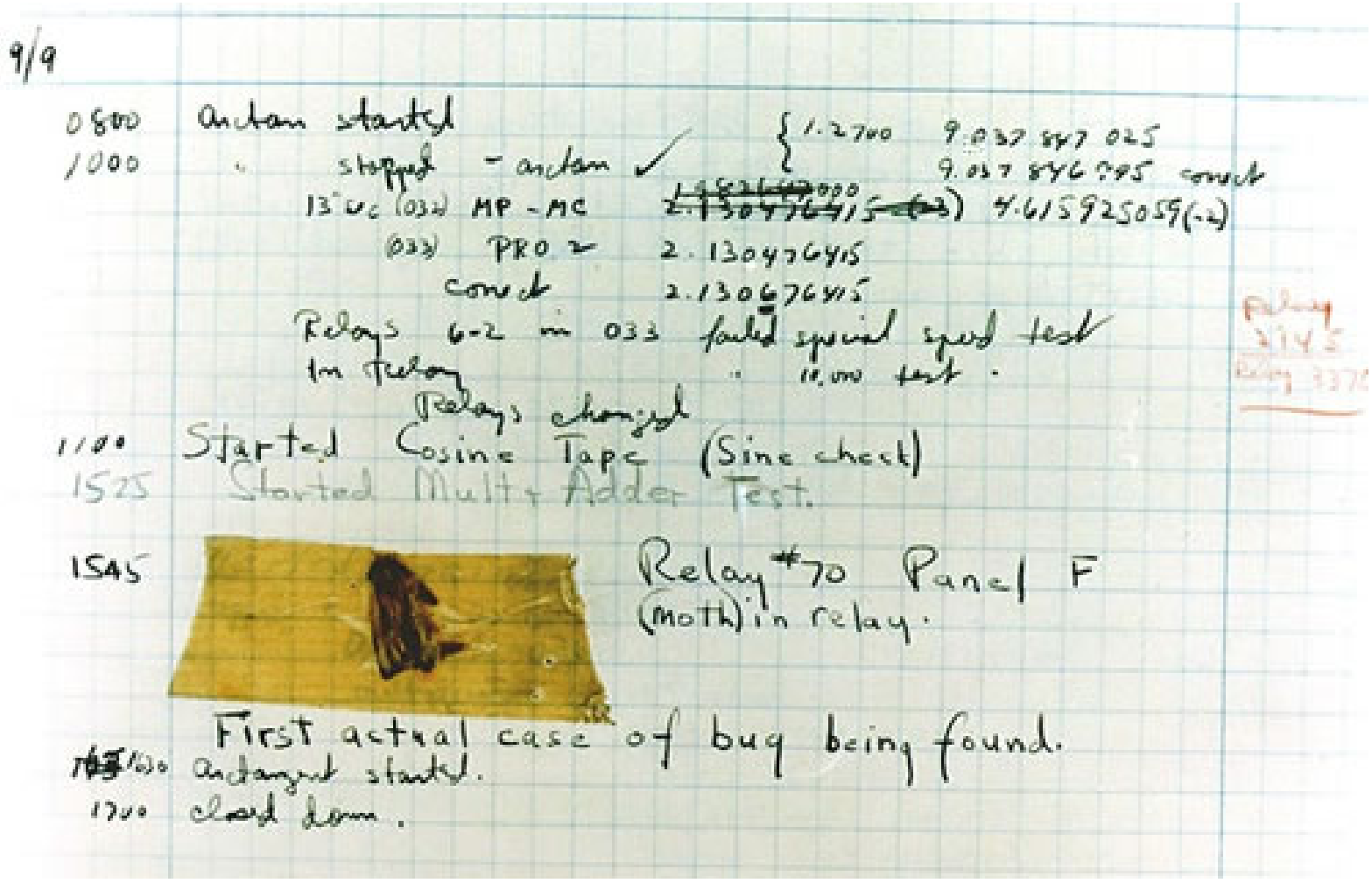
- Ce n'est pas parce que l'on peut démontrer qu'il existe une solution à un problème que cette solution est calculable e□cacement.
- C'est au programmeur de l'expliciter à l'aide du langage de programmation.

À quel niveau de détails doit-on expliciter la solution ?



- Autrefois, il y a bien longtemps, les programmes étaient écrits sur des fiches perforées.
- La manipulation de ces fiches était un travail de fourmi, long et fastidieux.
- Éviter les erreurs dans la réalisation d'un programme était alors très di□cile.
- Aujourd'hui, les outils de développement logiciel ont simplifié ce processus.

Comment éviter les erreurs ?



Le premier « bug ».

Comment éviter les erreurs ?

- Il y a différents types d'erreurs dans un programme.
- Les erreurs de fonctionnement apparaissent lorsque la description du calcul contient une opération illicite qui empêche l'obtention d'un résultat :
  - "`segmentation fault`" : écriture dans une zone mémoire non allouée ;
  - "`Uncaught exception`" : un cas exceptionnel n'est pas traité.

⇒ Il faut s'assurer que les opérations effectuées par la machine ont un sens.

- Lorsque le programme produit un résultat mais qu'il ne respecte pas la spécification, on parle d'erreur de correction.

⇒ Le calcul n'est pas le bon ! Il faut raisonner sur le programme pour comprendre ce qu'il calcule. Est-il facile de raisonner sur un programme ? Peut-on prouver qu'un programme vérifie une spécification ?

Résoudre la tension entre le « quoi ? » et le « comment ? »

- Pour faciliter le raisonnement sur les programmes, les langages de programmation réduisent la distance entre le « quoi ? » et le « comment ? » en

Introduisant des mécanismes calculatoires
de plus en plus en abstraits et éloignés des détails de la machine.

- Abstraire, c'est capturer ce qui est strictement utile à la programmation en mettant de côté certains détails d'implémentation non pertinents (relativement aux types de programmes écrits).
  La programmation est ainsi plus sûre.

Des langages de programmation plus ou moins abstraits

- Plus un langage de programmation fournit des mécanismes calculatoires abstraits des détails de fonctionnement de la machine et plus le langage est dit de haut-niveau.

Exemple : s'abstraire de l'ordre des calculs

- Langages fonctionnels :
  Un programme est un ensemble de fonctions.
- Langages logiques :
  Un programme est un ensemble de formules.
- Langages de programmation par contraintes :
  Un programme est un ensemble de contraintes à satisfaire.
- Langages orientés objets :
  Un programme est un ensemble d'objets qui collaborent en communiquant par envoi de messages.

Comment décrire le calcul ?

- Dès lors qu'ils s'affranchissent des opérations élémentaires de la machine qui les exécutent, les mécanismes des langages de programmation ne peuvent plus s'exprimer simplement en fonction de ces derniers.
- Il faut se doter d'une interprétation univoque et indépendante de chaque langage de programmation : une sémantique.

Comment définir une sémantique ?

- On peut donner la sémantique à un langage de plusieurs façons.
- Une sémantique dénotationnelle interprète un programme dans un espace mathématique (comme une fonction par exemple).

$$\begin{aligned} \text{fact} : \quad & \mathbb{N} \to \mathbb{N} \\ & n \mapsto n! \end{aligned}$$

- Une sémantique opérationnelle en donne une description calculatoire :

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n$$

- En suivant les règles données par la sémantique opérationnelle d'un langage de programmation $\mathcal{L}$, on peut écrire un programme, appelé interprète, qui permet d'exécuter tout programme écrit en $\mathcal{L}$.

Un exemple de sémantique opérationnelle

- Un langage de programmation est défini ainsi :

  - Un programme est une liste d'instructions.
  - Une instruction est l'une des deux formes suivantes :
    - `Avance N` où `N` est un entier représentant un nombre de pixel.
    - `Tourne N` où `N` est un entier représentant un angle en degré.

- On suppose un environnement d'exécution constitué d'un plan fini à deux dimensions, d'un position courante et d'une orientation courante.
- La sémantique opérationnelle pourrait alors être décrite ainsi :

  - Pour chaque instruction i d'un programme P :
    - Si $i \equiv$ "`Avance N`" alors tracer une ligne de longueur `N` depuis la position courante et en suivant la direction courante.
    - Si $i \equiv$ "`Tourne N`" alors la direction courante est augmentée de l'angle `N`.

$\Rightarrow$ Comment écrire un interprète pour ce langage ?

Qu'est-ce qui différencie un langage d'un autre ?

- Il y a une infinité de façon de représenter un calcul.
- Dans le cas précédent, un calcul est une succession de lignes tracées à l'écran.
- En C, un calcul est une transformation de l'état de la machine.
- Dans le cas d'un mélange chimique, un calcul est un ensemble de réactions.

$\Rightarrow$ Ce sont autant de modèles de calcul différents.

Une machine abstraite pour chaque modèle de calcul

- Une machine abstraite décrit l'environnement et les règles d'évaluation d'un modèle de calcul.
- Très peu de modèles possèdent une réalisation physique de leur machine abstraite (à l'exception du modèle de Von Neumann).
- On peut cependant émuler une machine abstraite à l'aide d'un autre programme.
- Un tel programme est appelé machine virtuelle.

Exemples de machines virtuelles

- La *Java Virtual Machine* (JVM) est une machine virtuelle implémentée par le progamme `java`. Le programme `ocamlrun` est la machine virtuelle du langage OCaml.
- Le programme *Qemu* émule un ordinateur IBM PC.
- On peut écrire un programme qui émule une machine de Turing.

Une limitation gênante, un compilateur pour la contrer

- L'émulation d'une machine abstraite ou l'utilisation d'un interprète introduisent une *inévitable inefficacité* : pour chaque opération X de la machine abstraite, il est nécessaire de décoder sa représentation et de l'émuler au fur et à mesure par des instructions de la machine réelle. Il y a donc une couche supplémentaire de traduction.
- Un compilateur répond à l'objectif de limitation de ces couches d'interprétation en *traduisant un programme écrit dans un langage $\mathcal{L}$ en un programme équivalent écrit dans un langage $\mathcal{L}$* et qui possède une réalisation e□cace de sa machine abstraite.

Exemples de compilation

- Le langage C se compile en code assembleur *i*386 qui est interprété (très) e□cacement par les processeurs d'architecture *Intel*.
- Le langage Java se compile vers un langage appelé code-octet Java qui est interprété par la machine virtuelle Java.
- Python est compilé vers un sous-ensemble de Python qui est interprété par la machine virtuelle de Python.

La spécification d'un compilateur

- Un compilateur attend des programmes écrits dans un certain langage source et qui ont un sens.
- La précondition d'un compilateur est donc une certaine propriété de bonne formation des programmes d'entrée (être syntaxiquement corrects, être bien typés, ...).
- La sortie d'un compilateur est un programme écrit dans le *langage cible*.
- On s'attend d'une part à ce que ce programme ait un sens mais, et c'est bien plus fort, à ce qu'il **ait un sens équivalent** au programme d'entrée.
- En résumé, la propriété fonctionnelle essentielle d'un compilateur est donc d'être **une traduction préservant la sémantique** des programmes.

Quelques mises au point...

- Un compilateur est une fonction qui traduit un programme écrit dans un langage source en un programme écrit dans un langage cible.
- Un interprète pour un langage L est une fonction qui évalue un programme écrit dans le langage L.
- En pratique, ces outils s'utilisent de multiples façons :
  - Le mode *batch* qui consiste à intégrer la fonction de compilation ou d'interprétation dans un programme et à l'utiliser sur des fichiers, appelés unités de compilation.
  - Le mode interactif qui consiste à intégrer la fonction de compilation et/ou d'interprétation dans une boucle interactive de façon à compiler et/ou interpréter les entrées de l'utilisateur au fur et à mesure de leur définition.

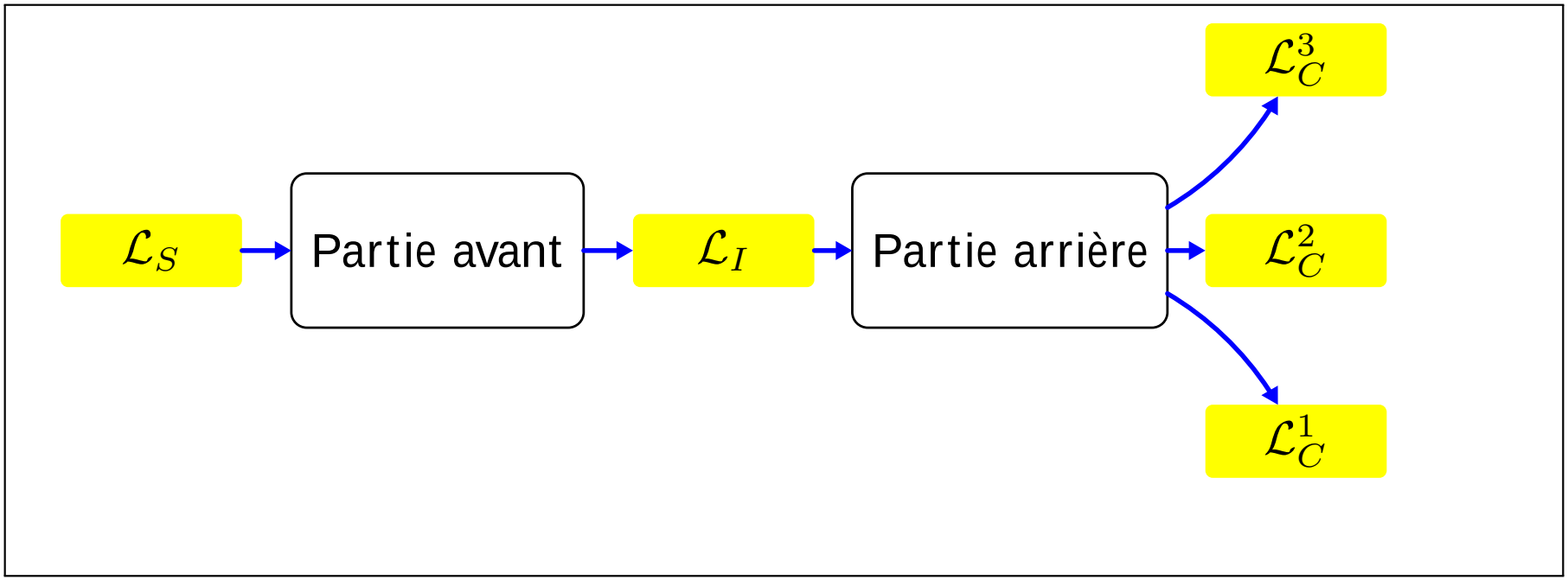
Exemples d'outils d'interprétation et de compilation

- Le programme `python` est un interprète en mode interactif quand il est utilisé sans argument, en mode *batch* si on lui donne un fichier `py`.
- Le programme `ocaml` est un compilateur vers la machine virtuelle `ocaml` et un interprète pour cette machine virtuelle dans une boucle interactive.
- Les programmes `gcc`, `ocamlc`, `ocamlopt`, `javac` sont des compilateurs en mode *batch*.

Quels sont les techniques d'implémentation utilisées en compilation ?

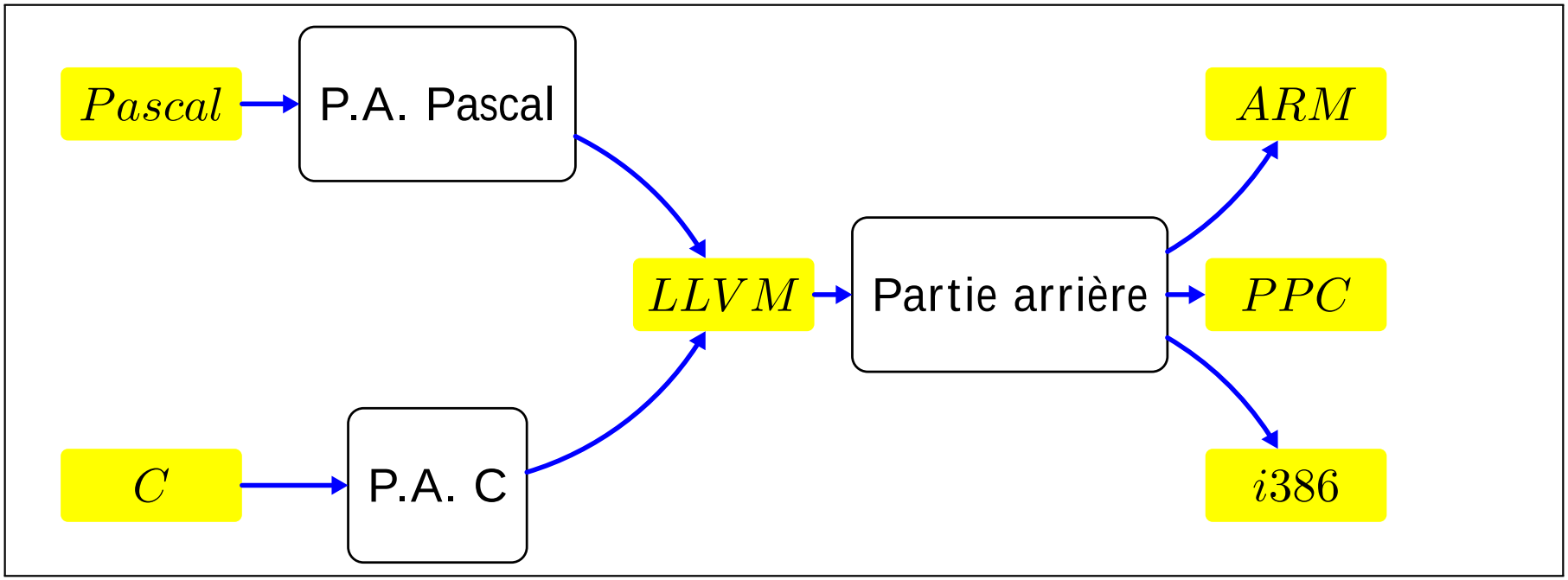
Architecture d'un compilateur (de loin)

Architecture d'un compilateur (d'un peu plus près)



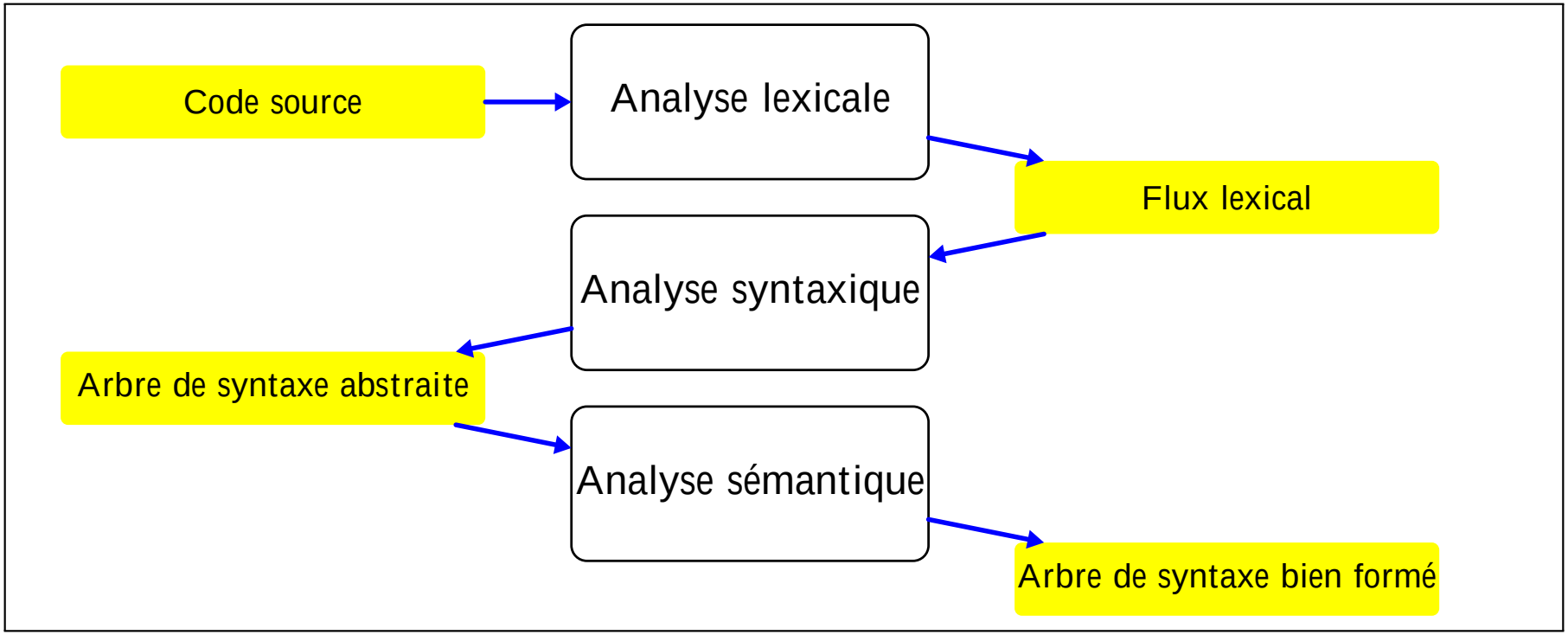
- Une architecture très courante consiste à séparer le compilateur en deux sous-compilateurs.
- La partie avant (*front-end*) traduit un programme du langage source $\mathcal{L}_S$ en un programme d'un langage intermédiaire $\mathcal{L}_I$.
- Ce programme de $\mathcal{L}_I$ est ensuite traduit par la partie arrière du compilateur en programmes de langages cibles di□érents.

Architecture d'un compilateur (d'un peu plus près)



- On peut ainsi réutiliser la partie arrière pour factoriser la conversion vers des cibles di□érentes.
- Le choix du langage intermédiaire est crucial.

L'architecture d'un compilateur : la partie avant



- Le cours de cette année porte essentiellement sur la partie avant.

Un micro-compilateur dans une boucle interactive

- Pour comprendre les différentes phases de la compilation, nous allons implémenter un micro-compilateur pour un langage de calcul arithmétique, que nous appelerons Marthe.

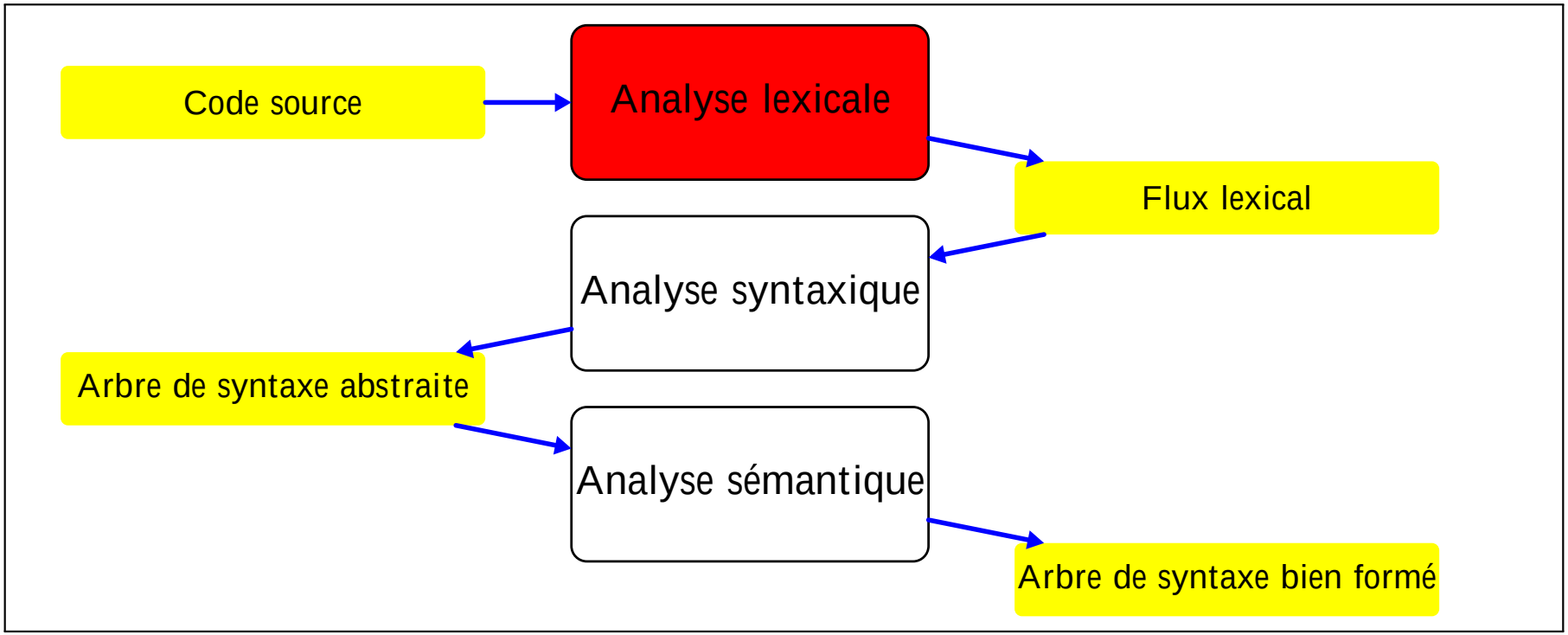
Structure d'une boucle interactive

Read - **E**val - **P**rint **L**oop

- On structure traditionnellement les boucles interactives à l'aide de quatre fonctions :
  - "Read" : on lit une chaîne représentant un programme dans un langage L.
  - "Eval" : on évalue cette chaîne si elle a du sens.
  - "Print" : on affiche le résultat.
  - "Loop" : on revient à la première étape.
- La fonction "Eval" peut se réaliser de di□érente façon :
  - grâce à un interprète du langage L ;
  - grâce à un compilateur du langage L vers un langage L' pour lequel on a un interprète.
  - avant l'exécution du programme, un *analyseur statique* peut vérifier qu'il va s'exécuter sans erreur.

Boucle interactive en O'Caml

```
let loop read eval print =
  let rec aux () =
    let entry = read () in
    let result = eval entry in
    print result ;
    aux ()
  in
  aux ()
```


Code source
Analyse lexicale
Flux lexical
Analyse syntaxique
Arbre de syntaxe abstraite
Analyse sémantique
Arbre de syntaxe bien formé

Un micro-compilateur : langage source

- Traditionnellement, le code source d'un programme est écrit dans un éditeur de texte et sauvegarder sous cette forme. Dans une boucle interactive, il peut aussi être saisi à l'aide d'un terminal.
- Dans tous les cas, cette entrée textuelle est une séquence de caractères (ASCII ou UNICODE).
- Voici quelques exemples de programmes que pourrait écrire un programmeur à destination de notre micro-compilateur de Marthe :

1. "7 * 3 + 7 * 3"
2. "sum (i, 1, 10, i * i)"
3. "sum (j, 1, 5, i + i / 2)"
4. "-3 / 0 + 1"

- Peut-on donner un sens à tous ces programmes ?

Un micro-compilateur : analyse lexicale

- On ne veut pas différencier les programmes suivants :

1. "7 * 3 + 7 * 3"
2. "7 * 3 + 7 * 3"
3. " 7 *3 +
   7 * 3"
4. "(* Ceci est un commentaire. *) 7 *3 + 7 * 3"

Un micro-compilateur : analyse lexicale

- L'analyse lexicale traduit une chaîne de caractères en un flux de lexèmes (*tokens* en anglais) : ce sont les unités lexicales qui ont un sens du point de vue du langage de programmation et qui interviennent dans la spécification de sa grammaire.
- Les programmes précédents sont tous transformés en :

  INT(7) ; STAR ; INT(3) ; PLUS ; INT(7) ; STAR ; INT(3)

  - "INT(k)" signifie "un entier valeur k".
  - "STAR" représente le symbole "*".
  - "PLUS" représente le symbole "+".
  - Les espaces, sauts de ligne et commentaires, quelque soit leur nombre, ont servi de séparateurs entre ces unités lexicales.

Un analyseur lexical en O'Caml

```
type token =
  | Int of int
  | Id of string
  | Sum
  | Plus
  | Star
  | Lparen
  | Rparen
  | Comma

exception LexingError of string
```

```
let lexer s =
  let rec number start i =
    let return () =
      (int_of_string (String.sub s start (i - start)), i)
    in
    if i ≥ String.length s then return () else match s.[i] with
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' →
        number start (i + 1)
      | _ →
        return ()
  in
  let rec identifier start i =
    let return () = (String.sub s start (i - start), i) in
    if i ≥ String.length s then return () else match s.[i] with
      | c when c ≥ 'α' ∧ c ≤ 'z' →
        identifier start (i + 1)
      | _ →
        return ()
  in
  let rec aux i =
    if i ≥ String.length s then [EOF] else match s.[i] with
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' →
        let (n, i) = number i i in
        Int n :: aux i
      | ' ' → aux (i + 1)
      | '×' → Star :: aux (i + 1)
      | '+' → Plus :: aux (i + 1)
      | '(' → Lparen :: aux (i + 1)
      | ')' → Rparen :: aux (i + 1)
      | c when c ≥ 'α' ∧ c ≤ 'z' →
        let (s, i) = identifier i i in
        (if s = "sum" then Sum else Id s) :: aux i
      | ',' → Comma :: aux (i + 1)
      | _ →
        raise (LexingError "Invalid character")
  in
  aux 0
```

Un micro-compilateur : analyse lexicale

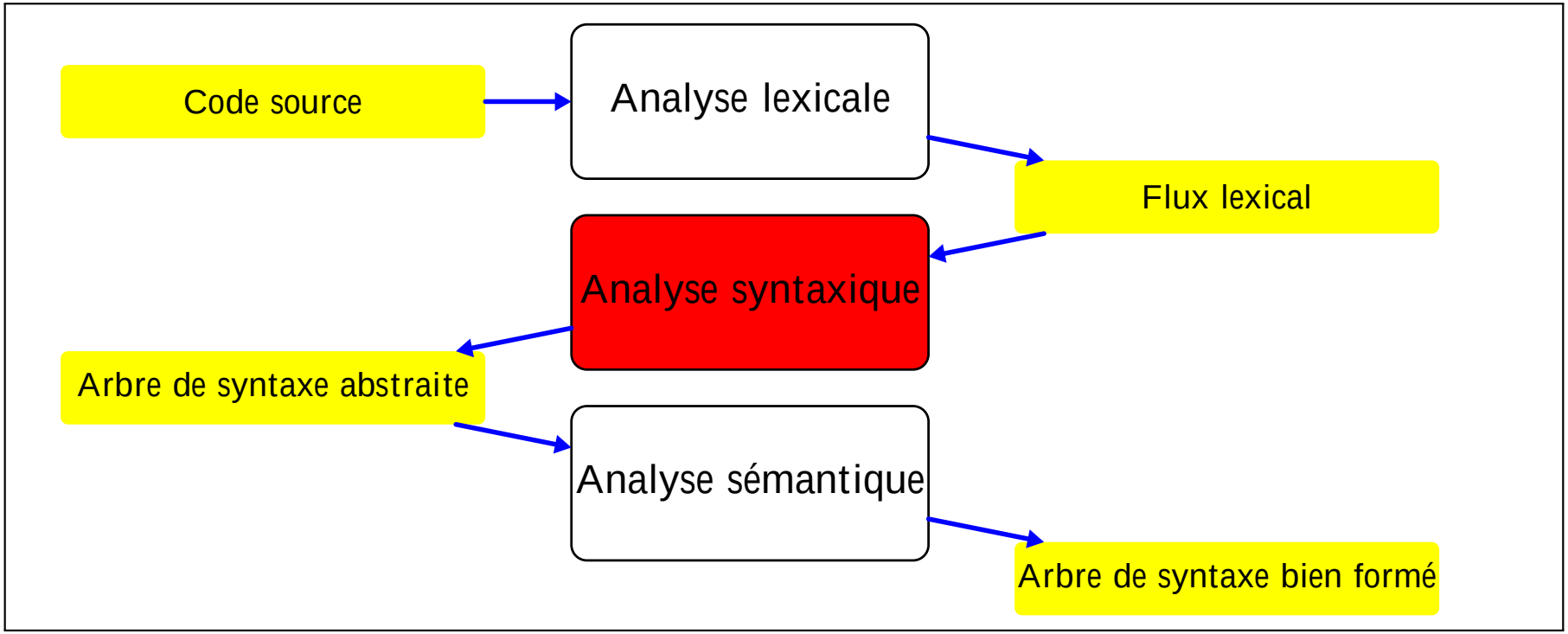
- L'analyse lexicale est implémentée à l'aide d'un *automate fini* (la plupart du temps).
- On décrit chaque sorte de lexème à l'aide d'une expression régulière.
- Exemples :
  - `[0-9]+` : les entiers naturels en base 10.
  - `[a-z]+` : les identifiants formés de lettres minuscules.
  - `(A|B)[a-z]*` : les identifiants qui commencent par un 'A' ou un 'B'.
- L'automate correspondant à cette expression régulière est calculé.
- On parcourt la chaîne de caractère du programme source.
- À partir du point courant de cette chaîne, on détermine quel est l'automate qui reconnaît la plus longue sous-chaîne pour savoir quel lexème produire.

Un micro-compilateur : analyse lexicale

- Le programme qui calcule les automates est un générateur d'analyseur lexical.
- En OCaml, on utilise `ocamllex`. En C, on utilise `flex`.
- Ces programmes sont des générateurs de code : il transforme une spécification d'analyseur lexical en un programme (OCaml ou C par exemple).
- Leur utilisation sera l'objet du prochain TD.

Un micro-compilateur : analyse lexicale

```
{
  type token = Int of int | Plus | Star
}

rule token = parse
  | ['0'-'9']+ { Int (int_of_string (Lexing.lexeme lexbuf)) }
  | "+" { Plus }
  | "×" { Star }
  | " " { token lexbuf }
  | _ { failwith "Unknown character." }

{
  let print = function
    | Int x → Printf.printf "Int(%d)" x
    | Plus → Printf.printf "Plus"
    | Star → Printf.printf "Star"

  let lexbuf = Lexing.from_string Sys.argv.(1)

  let rec loop () =
    try
      print (token lexbuf);
      Printf.printf "; ";
      loop ()
    with _ → ()

  let _ = loop ()
}
```


Code source
Analyse lexicale
Flux lexical
Analyse syntaxique
Arbre de syntaxe abstraite
Analyse sémantique
Arbre de syntaxe bien formé

Un micro-compilateur : analyse syntaxique

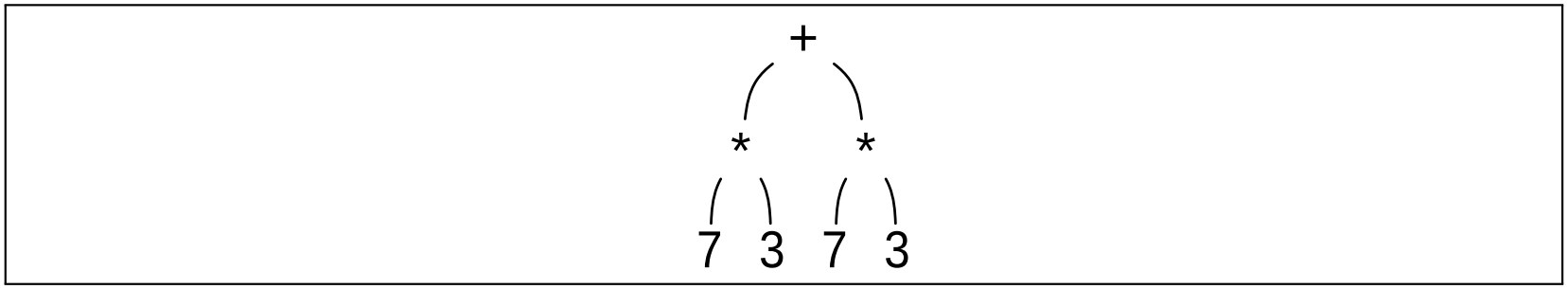
- Un flux de lexèmes a une structure linéaire qui n'est pas pratique pour interpréter le programme.
- À première vue, écrire un programme qui interprète correctement :

  INT(7) ; STAR ; INT(3) ; PLUS ; INT(7) ; STAR ; INT(3)

  . . . n'est pas aisé puisque l'on ne peut pas prendre une décision locale.
- En effet, il serait erroné d'effectuer l'addition avant d'avoir calculé le résultat de la multiplication $7 * 3$, à gauche et à droite.

Un micro-compilateur : analyse syntaxique

- On représente mieux une expression à l'aide d'une structure arborescente :



- Voici un type OCaml pour représenter cet arbre :

```
type e =
   | Int of int
   | Var of string
   | Plus of e × e
   | Mult of e × e
   | Div of e × e
   | Sub of e × e
   | Sum of string × e × e × e
```

- Cette représentation d'un programme est appelée arbre de syntaxe abstraite.
- Pour interpréter le nœud d'un arbre, il su□t de s'appuyer sur l'interprétation de ses sous-arbres.

Micro-compilateur : analyse syntaxique

- A priori, certaines séquences de lexèmes n'ont pas de structure associée :

  1. "7 * 3 + "
  2. "7 ** 3"
  3. "(1 + 3))"

- La grammaire du langage spécifie les séquences syntaxiquement correctes.
- Dans le cas de Marthe, la syntaxe des expressions pourrait être :

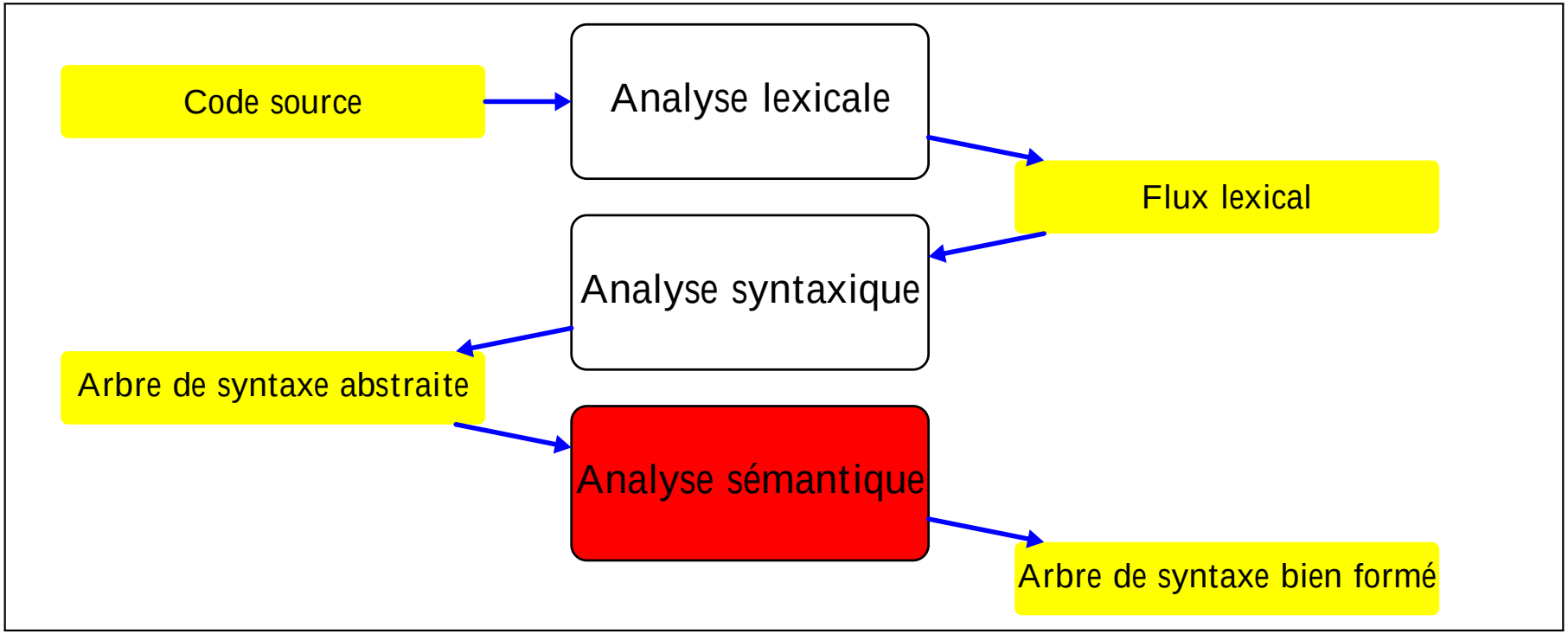
$$
\begin{array}{lcl}
e & ::= & \mathit{INT} \\
 & | & e\ \mathit{PLUS}\ e \\
 & | & e\ \mathit{STAR}\ e \\
 & | & \mathit{SUM}\ \mathit{LPAREN}\ \mathit{ID}\ \mathit{COMMA}\ e\ \mathit{COMMA}\ e\ \mathit{COMMA}\ e\ \mathit{RPAREN}
\end{array}
$$

- La notation utilisée ici s'appelle BNF, pour Backus-Naur Form.

Micro-compilateur : analyse syntaxique

- Le rôle de l'analyse syntaxique est double :
  1. Rejeter les programmes incorrects syntaxiquement.
  2. Produire un arbre de syntaxe abstraite dans le cas contraire.
- Les générateurs d'analyseurs syntaxiques transforment une spécification de grammaire en un programme exécutable qui réalise les deux opérations précédentes.

$\Rightarrow$ Nous étudierons les algorithmes sur lesquels ils s'appuient.


Code source
Analyse lexicale
Flux lexical
Analyse syntaxique
Arbre de syntaxe abstraite
Analyse sémantique
Arbre de syntaxe bien formé

Un analyseur syntaxique en O'Caml

```
let parse tokens =
  let (accept, current, next) =
    let tokens_stream = ref tokens in
    let next () =
      match !tokens_stream with
        | [] → raise (ParseError ("No more token", EOF))
        | tok :: tokens →
          tokens_stream := tokens
    in
    let current () =
      match !tokens_stream with
        | [] → assert false
        | tok :: _ →
          tok
    in
    let accept token =
      if (current () ≠ token) then
        raise (ParseError ("Unexpected token", token));
      next ()
    in
    (accept, current, next)
  in

  let rec phrase () =
    let e = expression () in
    accept EOF;
    e
```

```
and expression () =
  let e = term () in
  match current () with
    | EOF → e
    | Plus → next (); EPlus (e, expression ())
    | token → e

and term () =
  let t = factor () in
  match current () with
    | Star → next (); EMult (t, term ())
    | token → t

and factor () =
  match current () with
    | Lparen →
      next (); let e = expression () in accept Rparen; e
    | Sum →
      next ();
      accept Lparen;
      let id =
        match current () with
          | Id s → next (); s
          | token →
            raise (ParseError ("Expecting an identifier",
                               token))
      in
      accept Comma;
      let start = expression () in accept Comma;
      let stop = expression () in accept Comma;
      let body = expression () in accept Rparen;
      ESum (id, start, stop, body)
    | Id x → next ();EVar x
    | Int x → next (); EInt x
    | token →
      raise (ParseError ("Unexpected token", token))
in
phrase ()
```

Micro-compilateur : analyse sémantique

- On peut donner une sémantique à Marthe à l'aide de phrases françaises :
  - Pour calculer $e_1 + e_2$, calculer e_1, calculer e_2 et faire la somme des deux résultats.
  - Pour calculer $e_1 * e_2$, calculer e_1, calculer e_2 et faire le produit des deux résultats.
  - Pour calculer $sum(i, e1, e2, e3)$, calculer la valeur *start* de $e1$, calculer la valeur *stop* de $e2$, faire la somme de toutes valeurs de $e3$ pour i valant tous les entiers de de *start* à *stop*.
- La sémantique est ici opérationnelle et s'intéresse uniquement à la valeur issue de l'évaluation de l'expression.

$\Rightarrow$ Nous utiliserons un formalisme plus clair et plus concis pour définir les sémantiques opérationnelles des langages de programmation à base de règles d'inférence.

Micro-compilateur : analyse sémantique

- Malgré une syntaxe correcte, certains programmes n'ont pas de sens.
- Les deux premiers des programmes suivants calculent respectivement 42 et $\Sigma_{i=1}^{10} i^2$ tandis que les deux derniers n'ont pas d'évaluation.

1. "7 * 3 + 7 * 3"
2. "sum (i, 1, 10, i * i)"
3. "sum (j, 1, 5, i + i / 2)"
4. "-3 / 0 + 1"

- On aimerait rejeter le programme numéro 3 parce qu'il fait référence à une variable *i* indéfinie. On aimerait peut-être aussi alerter le programmeur de la non-utilisation de la variable *j*.
- On aimerait rejeter le programme numéro 4 parce qu'il divise par zéro.

$\Rightarrow$ Peut-on formuler une règle générale (calculable, menant à un algorithme) pour décider le rejet ou l'acceptation d'un programme ?

Micro-compilateur : analyse sémantique

- Règle 1 : “Pas de référence à des variables indéfinies”

  Soient un arbre de syntaxe P et $\mathcal{V}$, la liste des variables définies pour P.
  On suit un parcours préfixe de l'arbre et on applique les cas suivants :

  - *Si la racine de P est un nœud sum alors son premier sous-arbre est une variable i, la rajouter dans la liste $\mathcal{V}$ et continuer le parcours dans le dernier sous-arbre.*
  - *Si la racine de P est une variable i, vérifier que i est dans $\mathcal{V}$.*

Micro-compilateur : analyse sémantique

- Essai de règle 2 : “Pas de division par zéro”

  Soient un arbre de syntaxe P. On suit un parcours préfixe de l'arbre et on applique les cas suivants :

  - *Si la racine de P est un nœud “division” alors si le second sous-arbre **s'évalue en zéro**, on doit rejeter le programme.*

- Évaluer le programme que l'on compile n'est pas vraiment envisageable. D'abord, parce que ce n'est pas le rôle d'un compilateur mais celui d'un interprète. Ensuite, un programme écrit dans un langage généraliste peut ne pas terminer : si le programme ne termine pas alors le compilateur risque de ne jamais terminer lui non plus !

Micro-compilateur : analyse sémantique

- Nouvel essai de règle 2 : “Pas de division par zéro”

  Soient un arbre de syntaxe P. On suit un parcours préfixe de l'arbre et on applique les cas suivants :

  - *Si la racine de P est un nœud “division” et si le second sous-arbre est la constante 0 alors on rejette le programme.*

- Ce n'est toujours pas satisfaisant !
- Le compilateur ne va pas rejeter le programme “$1/(0+0)$” puisque “$0+0$” n'est pas syntaxiquement égal à 0.
- Si un programme est accepté, alors cela ne signifie pas qu'il ne divisera pas par 0.

Micro-compilateur : analyse sémantique

- Existe-t-il une règle qui accepte seulement des programmes qui ne divisent pas par zéro ?

$\Rightarrow$ On peut bannir la division ! Une autre idée ?

- Existe-t-il une règle qui accepte **uniquement les** programmes qui ne divisent pas par zéro (sans l'évaluer) ?

Micro-compilateur : analyse sémantique

- Le rôle de l'analyse sémantique est d'appliquer des règles pour fournir des garanties sur l'exécution du programme compilé.
- Dans notre exemple, la règle numéro 1 garantit qu'au cours du calcul, on fera toujours référence à des variables connues. Dans le cas de notre langage, ceci su□t pour s'assurer que tous les calculs aboutiront à un résultat entier si aucune division par zéro n'a lieu.
- Nous nous concentrerons sur des algorithmes (de typage) permettant d'obtenir ce type de propriétés de sûreté de l'évaluation.

Interprète

- À l'aide de la sémantique opérationnelle, écrire un interprète sur les arbres de syntaxes abstraites est très simple :

```
let rec eval variables = function
  | Int x → x
  | Var x → List.assoc x variables
  | Plus (e1, e2) → eval variables e1 + eval variables e2
  | Sum (x, e1, e2, e3) →
      let start = eval variables e1 in
      let stop = eval variables e2 in
      let accu = ref 0 in
      for i = start to stop do
        accu := !accu + eval ((x, i) :: variables) e3
      done;
      !accu
```

- On utilise l'analyse de motif (*pattern matching*) d'OCaml et la récursion pour écrire une fonction définie par induction sur l'arbre de syntaxe abstraite.
- La machine abstraite du langage est formée de la liste des valeurs courantes des variables et du terme à évaluer.

⇒ Peut-on supprimer cette couche d'interprétation en compilant le programme vers un modèle d'exécution plus rapide ?

Micro-compilateur : partie arrière

- Soit le langage de programmation Asm dans lequel un programme est une suite d'instructions de la forme suivante :
  - "**pushi** n" où n est un entier.
  - "**pushv** x" où x est une variable.
  - "**add**", "**sub**", "**mul**", "**div**".
  - "**for** $i = start$ **to** $stop$ **do** i_1 ; ... i_n ; **done**"
- La machine abstraite de ce langage est formé d'une pile contenant entiers et noms de variable, d'une liste d'instructions à exécuter et d'un pointeur indiquant l'instruction en cours d'exécution.
- On peut traduire un arbre de syntaxe de Marthe en un arbre de syntaxe de Asm : il suffit d'expliciter l'ordre dans lequel les calculs sont effectués et on stocke les résultats temporaires sur la pile.

Micro-compilateur : partie arrière

- Voici quelques exemples de traductions :
  - “2 + 40” $\Rightarrow$ “**pushi** 2 ; **pushi** 40 ; **add**”.
  - “3 * 7 + 3 * 7” $\Rightarrow$ “**pushi** 3 ; **pushi** 7 ; **mul** ; **pushi** 3 ; **pushi** 7 ; **mul** ; **add**”
- La structure du programme est de nouveau linéaire !
- Mais cette fois-ci, le programme est très facile à évaluer...

Micro-compilateur : partie arrière

- On peut écrire un interprète pour Asm.
- On peut aussi compiler Asm vers un autre langage avec un modèle d'exécution plus e□cace et ainsi de suite.

Pourquoi étudier la compilation ?

Pourquoi étudier la compilation ?

- La probabilité que l'on vous demande d'écrire un compilateur au cours de votre vie professionnelle est faible. [1]
- Pourquoi donc étudier des algorithmes spécialisés à ce domaine ou la théorie des langages ?

1. Cependant, les langages dédiés à un domaine spécifique (*Domain Specific Languages*, DSL) ont connu un important essor ces dernières années et l'implantation de ces derniers nécessitent l'écriture de compilateurs...

Pour devenir de meilleurs programmeurs

- La compréhension des mécanismes fondamentaux des langages de programmation a□ne votre regard sur les programmes que vous écrivez.
- Un langage de programmation est une somme de mécanismes calculatoires.
- Ces mécanismes sont partagés entre les di□érents langages.
- Connaître ces mécanismes, c'est se faciliter l'apprentissage de nouveaux langages de programmation.
- Les comprendre en profondeur rend possible un raisonnement rigoureux sur les programmes.

$\Rightarrow$ Le cours de compilation est avant tout un cours de programmation.

Pour faire croître sa boîte à outils

- Si l'écriture de compilateur n'est pas une activité quotidienne pour la plupart des développeurs, l'utilisation et l'écriture d'outils d'analyse, de gestion et de génération de code sont très courantes.

$\Rightarrow$ Une partie des algorithmes de la compilation sont réutilisables dans ce contexte.

- Beaucoup de problèmes se comprennent mieux quand on s'est donné un langage pour les spécifier.
- Les langages sont partout : un format de fichier est un langage, une requête à un système s'écrit dans un langage, les traces des protocoles réseaux forment elles-aussi un langage...

$\Rightarrow$ La théorie des langages va vous donner les outils scientifiques pour les spécifier et implémenter rigoureusement.

Pour devenir de meilleurs informaticiens

- La compilation est un sujet transversal de l'informatique.
- Pour écrire un compilateur, on sollicite ses connaissances :
  - des automates ;
  - des algorithmes travaillant sur les arbres et sur les graphes ;
  - des architectures des ordinateurs ;
  - de théorie des langages ;
  - de génie logiciel.

La compilation dans le cursus de Paris Diderot

- L3 : Introduction à la compilation, Machines virtuelles, Programmation Fonctionnelle.
- M1 : Compilation, Sémantique, Génie logiciel, P.O. avancée, Programmation Fonctionnelle Avancée.
- M2 (LP) : Compilation avancée, P.O concepts avancés, Programmation Comparée.
- M2 (MPRI) : Tous les cours liés à la sémantique des langages de programmation.

Synthèse

Conclusion de ce premier cours introductif

- Le compilateur permet à la programmation de s'abstraire des détails de la machine en traduisant un langage source, de haut-niveau, en un langage cible, de plus bas niveau et qui possède un modèle d'exécution plus e□cace.
- Un compilateur est généralement la composée de plusieurs traductions.
- Nous nous intéresserons à l'analyse lexicale et syntaxique au prochain cours.

Plan du cours

- Analyse syntaxique du cours 2 au cours 6 :
  - Cours 2 : Introduction
  - Cours 3 : Analyse descendante
  - Cours 4 et 5 : Analyse ascendante
  - Cours 6 : Analyse syntaxique en pratique et révision
  - Partie 1 du projet : Un analyseur syntaxique à partir d'une spécification.
- Sémantique :
  - Cours 7 : De la syntaxe à la sémantique
  - Cours 8 : La liaison des variables
  - Cours 9 : L'état
  - Cours 10 et 11 : Les procédures
  - Cours 12 : Les fonctions de première classe
  - Cours 13 : Compilation des fonctions de première classe
  - Partie 2 du projet : Un interprète et un typeur pour le langage de la partie 1.

Fonctionnement du cours

- Le cours a lieu le vendredi de 10h30 à 12h30 en amphi 6C.

Venez en cours !

- Les travaux dirigés :
  - Le lundi de 16h à 18h, Thibaut Balabonski en 478F et 548C.
  - Le mercredi de 10h30 à 12h30, Matthias Puech en 470E et 548C.
  - Le jeudi de 9h00 à 11h00, Mihaela Sihireanu en 470E et 554C.
- Modalités d'évaluation :
  - 1ère session : 40% projet + 60% examen
  - 2ème session : 40% projet + 60% examen
  - Le projet est **obligatoire**.

Comment travailler le cours ?

- Connaître le cours :
  - La page du cours, contenant les ressources comme les transparents, les programmes écrits en cours ou encore les références bibliographiques : http://www.pps.jussieu.fr/~yrg/compil
- S'exercer :
  - Les travaux dirigés.
  - Des exercices seront donnés dans les notes de cours.
  - Des exercices de programmation, non obligatoires, vous seront proposés.

QCM : Question 1

Un compilateur a pour rôle :

a. de traduire un programme en assembleur.
b. de vérifier qu'un programme termine.
c. de traduire un programme dans langage dans un autre langage.
d. d'évaluer n'importe quel programme d'un certain langage.

QCM : Question 2

Un langage de programmation de haut-niveau :

a. est nécessairement interprété.
b. produit, une fois compilé, des programmes systématiquement plus lents que les programmes écrits dans des langages de bas-niveau.
c. aide à éviter certaines erreurs de programmation.
d. complique le raisonnement sur les programmes.

QCM : Question 3

La partie avant d'un compilateur :

a. comprend une phase d'analyse lexicale.
b. comprend une phase d'analyse syntaxique.
c. comprend une phase d'analyse sémantique.
d. comprend une phase de génération de code.
e. doit être réécrite pour chaque processeur.
f. doit être réécrite pour chaque langage.

QCM : Question 4

Laquelle de ces structures de données est la plus pratique pour représenter et évaluer une expression arithmétique :

a. une chaîne de caractère en notation infixe sans parenthésage explicite.
b. une chaîne de caractère en notation infixe avec parenthésage explicite.
c. une chaîne de caractère en notation postfixe.
d. un arbre de syntaxe abstraite.

QCM : Question 5

On peut écrire un algorithme qui a pour entrée le code source d'un programme P et détermine :

a. si P termine.
b. si P fera "*segmentation fault*".
c. si P fera une division par zéro.
d. si P peut faire une division par zéro.