

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

29 janvier 2010

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 ?

- ▶ Les conséquences d'une telle commande peuvent être multiples.
- ▶ 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 procédés utilisés 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 efficacement, par le calcul.

Résoudre des problèmes ?

- ▶ Il existe de nombreux types de problèmes : certains sont simples à résoudre et d'autres sont plus difficiles, 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

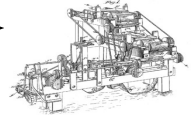
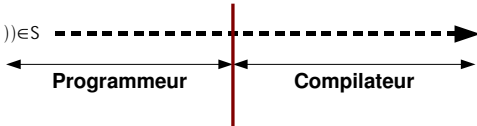
- ▶ Ce n'est pas parce que l'on peut démontrer qu'il existe une solution à un problème que cette solution est calculable efficacement.
- ▶ C'est au programmeur de l'explicitier à l'aide du langage de programmation.

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

Spécification du problème

Résolution effective par la machine

$\exists F. \forall I. (I, F(I)) \in S$



Langage de programmation

- ▶ 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.
- ▶ L'erreur dans la réalisation de la solution était alors très difficile à éviter.
- ▶ Aujourd'hui, les outils de développement logiciel ont simplifié ce processus.

Comment éviter les erreurs ?

9/9

0800 Antam started
1000 " stopped - antam ✓
1300 (032) MP-MC 1.2700 9.037 847 025
(033) PRO 2 2.130476415 9.037 846 995 correct
correct 2.130676415

Relays 6-2 in 033 failed special speed test
in relay 11.00 test.

Relay
2145
Relay 3370

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.
1630 Antam started.
1700 closed down.

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 erreur 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.
- ⇒ Le déroulement du calcul est problématique. 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**.

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

- ▶ Les langages de programmation visent à apaiser la tension 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 utile à la programmation, au sens où on la rend plus compréhensible, plus économique et plus sûre.

Des langages de programmation plus ou moins abstraits

- ▶ Plus un langage de programmation fournit des mécanismes calculatoires abstraits des détails 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 réagissant à des messages.
- ⇒ Ce sont des langages **déclaratifs**.
- ⇒ Les langages **impératifs** ne s'abstraient pas de l'ordre des calculs.

Comment décrire le calcul ?

- ▶ Dès lors qu'ils s'affranchissent des mécanismes basiques 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 ?

- ▶ Les définitions des sémantiques des langages de programmation peuvent prendre de multiples formes.
- ▶ Une sémantique **dénotationnelle** interprète un programme dans un espace mathématique (comme une fonction par exemple).

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

6 grammar

Un exemple de sémantique opérationnelle

- ▶ Un langage de programmation est défini ainsi :

- ▶ Un programme est une liste d'instructions.
- ▶ Une instruction est de 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 et d'un point courant et d'une orientation courante.
- ▶ La sémantique opérationnelle pourrait alors être décrite ainsi :

- ▶ Pour chaque instruction i d'un programme P :
 - ▶ Si $i \equiv \text{"Avance } N"$ alors tracer une ligne du point courant de taille N et en suivant la direction courante.
 - ▶ Si $i \equiv \text{"Tourne } N"$ alors la direction courante est augmentée de l'angle N .

⇒ 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.
 - ▶ Dans le cas d'un programme 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.
- ⇒ Ce sont autant de **modèles de calcul** différents.

Une machine abstraite pour chaque modèle de calcul

- ▶ Une **machine abstraite** décrit l'environnement 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 programme `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 *inévitabile inefficacité*.
- ▶ 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 efficace de sa machine abstraite.

Exemples de compilation

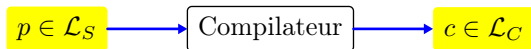
- ▶ Le langage C se compile en code assembleur *i386* qui est interprété (très) efficacement par les processeurs d'architecture *Intel*.
- ▶ Le langage JAVA se compile vers un langage appelé code-octet JAVA qui est interprété par la machine virtuelle JAVA.
- ▶ PYTHON est compilé vers un sous-ensemble de PYTHON qui est interprété par la machine virtuelle de PYTHON.

La spécification d'un compilateur

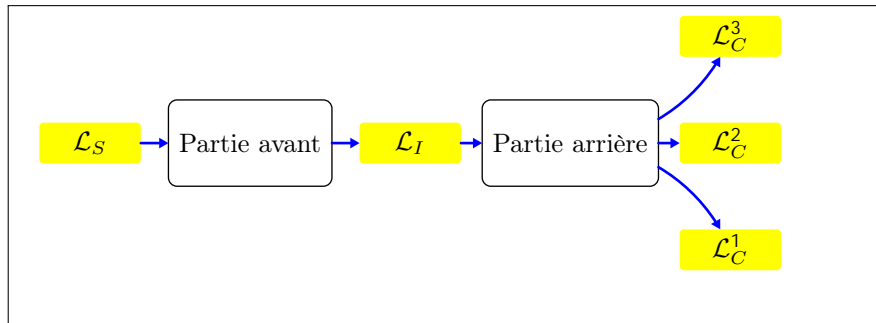
- ▶ Un compilateur attend des programmes écrits dans un certain **langage source** et qui ont un sens.
- ▶ La précondition d'un compilateur est donc une certaine propriété de bonne formation des programmes d'entrée (être syntaxiquement corrects, être bien typés, ...).
- ▶ La sortie d'un compilateur est 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.
- ▶

Quels sont les procédés utilisés par la compilation ?

Architecture d'un compilateur (de loin)

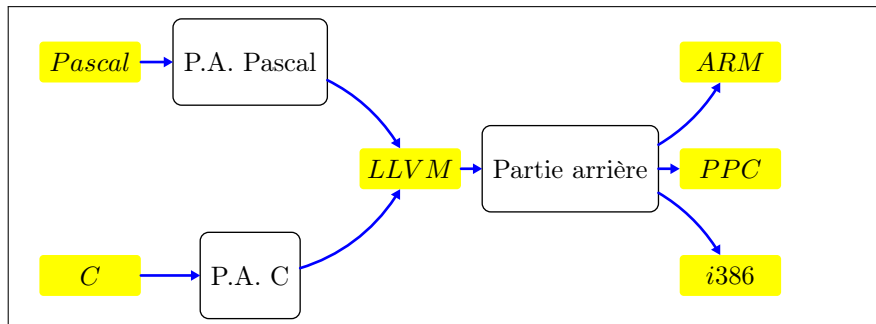


Architecture d'un compilateur (d'un peu plus près)



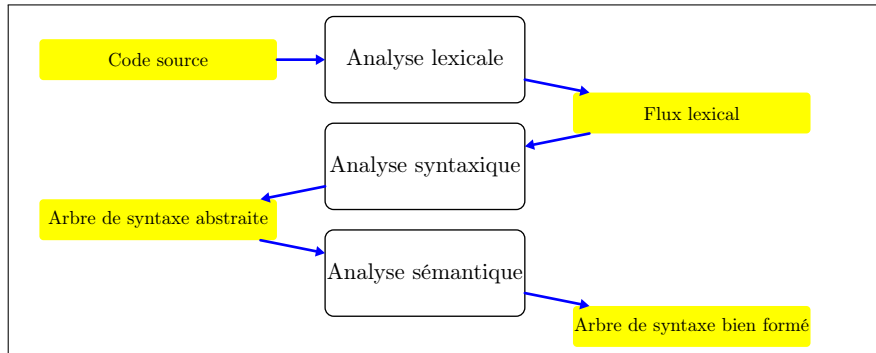
- ▶ Une architecture très courante consiste à séparer le compilateur en deux sous-compilateurs.
- ▶ La **partie avant** (*front-end*) traduit un programme du langage source \mathcal{L}_S en un programme d'un langage intermédiaire \mathcal{L}_I .
- ▶ Ce programme de \mathcal{L}_I est ensuite traduit par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

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



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

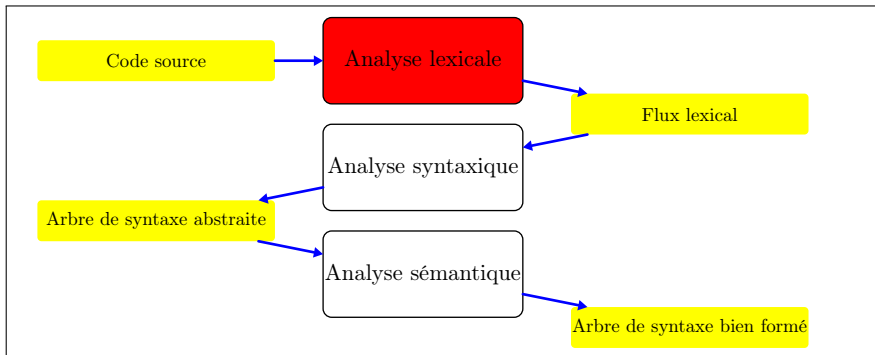
L'architecture d'un compilateur : la partie avant



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

Un micro-compileur

- Pour comprendre les différentes phases de la compilation, observons leurs entrées et les productions sur l'exemple d'un micro-compileur pour un langage de calcul arithmétique, que nous appellerons MARTHE.



Un micro-compileur : 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-compileur 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-compileur : 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 micro-compileur : 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-compileur : 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-compileur : analyse lexicale

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

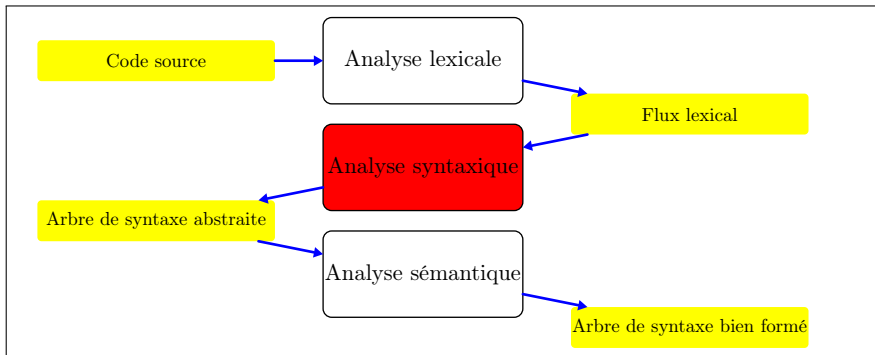
rule token = parse
| ['0'-'9']+ { Int (int_of_string (Lexing.lexeme lexbuf)) }
| "+" { Plus }
| "×" { Star }
| " " { token lexbuf }
| _ { failwith "Unknown character." }

{
  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 ()
}
```



Un micro-compileur : 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 :

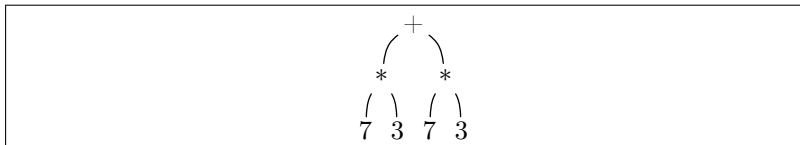
`INT(7); STAR; INT(3); PLUS; INT(7); STAR; INT(3)`

... 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 calculer le résultat de la multiplication $7 * 3$, à gauche et à droite.

Un micro-compileur : 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 suffit de s'appuyer sur l'interprétation de ses sous-arbres.

Micro-compileur : analyse syntaxique

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

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

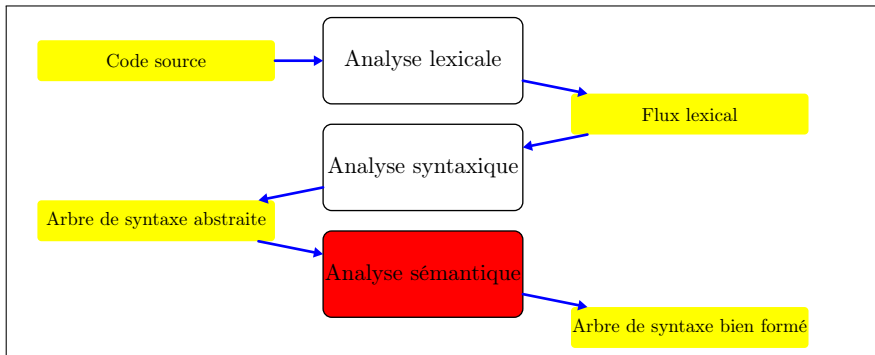
- ▶ La grammaire du langage spécifie les séquences syntaxiquement correctes.
- ▶ Dans le cas de MARTHE, la syntaxe des expressions pourrait être :

```
e ::= INT
    | e PLUS e
    | e STAR e
    | SUM LPAREN ID COMMA e COMMA e COMMA e RPAREN
```

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

Micro-compileur : 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.
- ⇒ Nous étudierons les algorithmes sur lesquels ils s'appuient.



Micro-compileur : 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*.
- ⇒ Nous utiliserons un **formalisme** plus clair et plus concis pour définir les sémantiques opérationnelles des langages de programmation.

Micro-compileur : analyse sémantique

- ▶ Malgré leur syntaxe correcte, certains programmes n'ont pas de sens.
- ▶ Les deux premiers programmes calculent respectivement 42 et $\sum_{i=1}^{10} i^2$ 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 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.
- ⇒ Dans ces deux cas, peut-on formuler pour chacun une règle générale (calculable) de rejet ?

Micro-compileur : 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-compileur : analyse sémantique

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

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 “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 envisageable. D'abord, parce que ce n'est pas le rôle d'un compilateur mais celui d'un interprète. Ensuite, si le programme ne termine pas alors le compilateur risque de ne jamais terminer lui non plus.

Micro-compileur : analyse sémantique

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

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 “division” alors 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-compileur : analyse sémantique

- ▶ Existe-t-il une règle qui accepte seulement des programmes qui ne divisent pas par zéro ?
- ⇒ On peut bannir la division ! Une autre idée ?
- ▶ Existe-t-il une règle qui accepte **uniquement les** programmes qui ne divisent pas par zéro ?

Micro-compileur : 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 suffit 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-compileur : 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, 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'explicitier 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 :
 - ▶ $"2 + 40" \Rightarrow \text{"pushi 2 ; pushi 40 ; add"}$.
 - ▶ $"3 * 7 + 3 * 7" \Rightarrow \text{"pushi 3 ; pushi 7 ; mul ; pushi 3 ; pushi 7 ; mul ; add"}$
- ▶ La structure du programme est de nouveau linéaire !
- ▶ Mais cette fois-ci, le programme est très facile à évaluer...

Micro-compileur : partie arrière

- ▶ On peut écrire un interprète pour Asm.
- ▶ On peut aussi compiler Asm vers un autre langage avec un modèle d'exécution plus efficace 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 ?

¹Cependant, 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 affine votre regard sur les programmes que vous écrivez.
 - ▶ Un langage de programmation est une somme de mécanismes calculatoires.
 - ▶ Ces mécanismes sont partagés entre les différents langages.
 - ▶ Connaître ces mécanismes, c'est se faciliter l'apprentissage de nouveaux langages de programmation.
- ⇒ Le cours de compilation est avant tout un cours de **programmation**.

Pour devenir le meilleurs développeurs

- ▶ 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.
- ⇒ 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. . .

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 : Compilation avancée, P.O concepts avancés, Programmation Comparée.

Synthèse

Conclusion de ce premier cours

- ▶ 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 efficace.
- ▶ Un compilateur est généralement la composée de plusieurs traductions.
- ▶ Nous nous intéresserons à l'analyse lexicale et syntaxique au prochain cours.

QCM : Question 1

Un compilateur a pour rôle :

- ☐ de traduire un programme en assembleur.
- ☐ de vérifier qu'un programme termine.
- ☐ de compiler efficacement un langage.
- ☐ d'évaluer n'importe quel programme d'un certain langage.

QCM : Question 2

Un langage de programmation de haut-niveau :

- ☐ est plus compliqué à utiliser qu'un langage de bas-niveau.
- ☐ produit des programmes plus lents que les programmes écrits dans des langages de bas-niveau.
- ☐ aide à éviter certaines erreurs de programmation.
- ☐ n'a pas une sémantique très claire.

QCM : Question 3

La partie avant d'un compilateur :

- ☐ comprend une phase d'analyse lexicale.
- ☐ comprend une phase d'analyse syntaxique.
- ☐ comprend une phase d'analyse sémantique.
- ☐ comprend une phase de génération de code.
- ☐ doit être réécrite pour chaque processeur.
- ☐ doit être réécrite pour chaque langage.

QCM : Question 4

Les structures de données suivantes sont pratiques pour représenter et évaluer une expression arithmétique :

- ☐ une chaîne de caractère en notation infixe sans parenthésage explicite.
- ☐ une chaîne de caractère en notation infixe avec parenthésage explicite.
- ☐ une chaîne de caractère en notation postfixe.
- ☐ un arbre de syntaxe abstraite.